Functional Database Query Languages as Typed Lambda Calculi of Fixed Order

(Extended Abstract)

Gerd G. Hillebrand[*†]
Brown University
ggh@cs.brown.edu

Paris C. Kanellakis[*]
Brown University
pck@cs.brown.edu

Abstract

We present a functional framework for database query languages, which is analogous to the conventional logical framework of first-order and fixpoint formulas over finite structures. We use atomic constants of order 0, equality among these constants, variables, application, lambda abstraction, and `let` abstraction; all typed using fixed order (≤ 5) functionalities. In this framework, proposed in [21] for arbitrary order functionalities, queries and databases are both typed lambda terms, evaluation is by reduction, and the main programming technique is list iteration. We define two families of languages: $\mathrm{TLI}_i^=$ or simply-typed list iteration of order $i+3$ with equality, and $\mathrm{MLI}_i^=$ or ML-typed list iteration of order $i+3$ with equality; we use $i+3$ since our list representation of databases requires at least order 3. We show that: FO-queries $\subseteq$ $\mathrm{TLI}_0^= \subseteq \mathrm{MLI}_0^= \subseteq$ LOGSPACE-queries $\subseteq$ $\mathrm{TLI}_1^= = \mathrm{MLI}_1^=$ = PTIME-queries $\subseteq \mathrm{TLI}_2$, where equality is no longer a primitive in TLI_2. We also show that ML type inference, restricted to fixed order, is polynomial in the size of the program typed. Since programming by using low order functionalities and type inference is common in functional languages, our results indicate that such programs suffice for expressing efficient computations and that their ML-types can be efficiently inferred.

1 Introduction

Motivation and Background: The logical framework of first-order and fixpoint formulas over finite structures has been the principal vehicle of theoretical research in database query languages; see [17, 18, 12, 13] for some of its earlier formulations. This framework has greatly influenced the design and analysis of *relational* and *complex-object database query languages* and has facilitated the integration of *logic programming* techniques in databases. The main motivation has been that common relational database queries are expressible in *relational calculus/algebra* [17], *Datalog*$^\neg$ and various *fixpoint logics* [4, 5, 29, 13, 14]. Most importantly, as shown in [23, 38], every PTIME query can be expressed using Datalog$^\neg$ on ordered structures; and, as shown in [4], it suffices to use Datalog$^\neg$ syntax under a variety of semantics to express various fixpoint logics. In addition, extensions have been proposed to this framework to manipulate complex-object databases, based on high-order formulas over finite structures, e.g., [2, 1]; see [3] for a short overview.

Despite the success of logical frameworks, it is not clear how to use them for the description and manipulation of *object-oriented databases*. *Functional programming*, with its emphasis on abstraction and on data types, might provide more insight into object-oriented database problems. There is a growing body of work on functional query languages, from the early FQL language of [11] to the more recent work on structural recursion as a query language [8, 10, 9, 25, 39]. In this context, it is natural to ask: "Is there a functional analog of the logical framework of first-order and fixpoint formulas over finite structures?" In [21] we partly answered this question by computing on finite structures with the typed λ-calculus. In this paper, we continue our investigation with a focus on fixed order fragments of the typed λ-calculus, where *order* is a measure of the nesting of type functionalities. We show these fragments are functional analogs of relational calculus/algebra and fixpoint characterizations of PTIME.

The *simply typed λ-calculus* [15] (*typed λ-calculus* or TLC for short) with its syntax and beta-reduction strategies can be viewed as a framework for database query languages which is between the declarative calculi and the procedural algebras. We use the "Curry view" of TLC without type annotations and infer *monomorphic* or simple types. We also use $\mathrm{TLC}^=$, the typed λ-calculus with atomic constants and an equality on them, and the associated delta-reduction

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

†Currently visiting INRIA Rocquencourt, France

SIGMOD/PODS 94 - 5/94 Minneapolis, Minnesota USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

of [15]. By adding `let`-polymorphism to TLC, Milner's ML language [34, 35] combines the convenience of type inference and the flexibility of polymorphism. So we also consider ML-typings. We refer to Section 2 for the necessary background in TLC (Section 2.1), ML (Section 2.2), and list iteration (Section 2.3).

The expressive power of TLC was originally analyzed in terms of computations on simply typed Church numerals (see, e.g., [6, 19, 36]). Unfortunately, the simply typed Church numeral input-output convention imposes severe limitations on expressive power. Only a fragment of PTIME is expressible this way (i.e., the extended polynomials). This does not illustrate the full capabilities of TLC. That more expressive power is possible follows from the fact that provably hard decision problems can be embedded in TLC, see [37, 33], and that different typings allow exponentiation [19].

One way of expressing all of PTIME, while avoiding the anomalies associated with representations over Church numerals was recently demonstrated by Leivant and Marion [31]. By augmenting the simply typed lambda calculus with a pairing operator and a "bottom tier" consisting of the free algebra of words over $\{0,1\}$ with associated constructor, destructor, and discriminator functions, they obtained various calculi in which there exist simple characterizations of PTIME. (Since Cobham's early work there have been a number of interesting *functional* characterizations of PTIME, e.g., [16, 20, 7]). In summary, to exhibit the power of TLC one must add features as in [31] and/or modify the input-output conventions.

In [21] we re-examined the expressive power of the typed λ-calculus, but over appropriately typed encodings of finite structures. We examined both the "pure" TLC and the "impure" TLC$^=$ and we obtained the following results: (1) TLC expresses exactly the elementary queries and, thus, is a functional language for the complex-object queries of [1]. (2) Every PTIME-query can be embedded in TLC so that its evaluation can be performed with a PTIME reduction strategy. (3) Every PTIME-query can be embedded in TLC$^=$, where the order of type functionalities is 4, so that its evaluation can be performed with a PTIME reduction strategy.

In this paper we analyze fixed order fragments of TLC and TLC$^=$. More specifically we use: atomic constants of order 0, equality among these atomic constants, variables, application, lambda abstraction, and `let` abstraction; all typed using at most order 5 functionalities. In this framework queries and databases are both typed lambda terms, evaluation is by reduction, and the main programming technique is list iteration. We define two families of languages: $\mathrm{TLI}_i^=$ or simply-typed list iteration of order $i+3$ with equality, and $\mathrm{MLI}_i^=$ or ML-typed list iteration of order $i+3$ with equality (we use $i+3$ since our list representation of databases requires at least order 3). Our input-output conventions are detailed in Section 3, for inputs (Section 3.1) and for queries (Section 3.2). We assume knowledge of the logical database framework.

Contributions: Our new results are detailed in Sections 4–7 and are as follows:

(1) In Section 4.1 we show that: FO-queries $\subseteq \mathrm{TLI}_0^= \subseteq \mathrm{MLI}_0^=$. These proofs are variants of those in [21], but on encodings that are more economical in order of functionality. We also show that by varying the typing of equality (but not its order) it is possible to express Parity, Majority and other non-FO queries. In Section 4.2 we briefly review the embedding of PTIME-queries in $\mathrm{TLI}_1^=$ of [21] and illustrate the use of types. For all these programs there are PTIME reduction strategies.

(2) In Section 5 we investigate the flexibility of ML-typing. We show that for fixed order functionalities ML-type inference is PTIME in the size of programs. In general, type inference is EXPTIME-complete in the size of programs [26, 27]. Thus, in our MLI languages type inference is provably efficient. These languages do simplify our calculations. For example, PTIME-queries $\subseteq \mathrm{MLI}_1^=$ is provable without any of the "type laundering" techniques of [21].

(3) In Section 6 we present the main analytic results of this paper. These are upper bounds on the expressibility of the TLI and MLI languages for $i = 0, 1$. To show these upper bounds we have to reason based on our input-output conventions. More specifically we prove that: $\mathrm{TLI}_1^= \subseteq \mathrm{MLI}_1^=$ $\subseteq$ PTIME-queries and $\mathrm{TLI}_0^= \subseteq \mathrm{MLI}_0^= \subseteq$ LOGSPACE-queries. These proofs involve an analysis of the structure of programs (Section 6.1) and an evaluator of programs (Section 6.2), which uses reduction plus specialized data structures. One consequence of this analysis is a functional characterization of PTIME that differs from those of [16, 20, 7, 31] in the sense of having the fewest additions to TLC—just equality over atomic constants.

(4) In Section 7 we show that every PTIME-query can be embedded in TLI_2, or TLC where the order of type functionalities is 5, so that its evaluation can be performed with a PTIME reduction strategy. This improves on [21] since it removes equality and still uses fixed order.

Finally, we would like to note that our analysis (except for the ML type inference) is for terms of order 5 or less. Beyond order 5 we believe (although we have not worked out the details here) that it should be possible to combine our basic machinery with the reductions of [28, 22, 30] to express various exponential time and space classes.

We close with some open questions in Section 8.

2 Typed Lambda Calculus Programs

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

TLC: 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*. For example, α is a type, as are $(\alpha \to \beta)$ and $(\alpha \to (\alpha \to \alpha))$. TLC λ-*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 by *well-typedness*. By standard convention, the type $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$ and the λ-term 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 $\Gamma[x{:}\sigma]$ is the function Γ' augmenting or updating Γ with $\Gamma'(x) = \sigma$:

$$\text{(VAR)} \qquad \frac{\Gamma(x) = \sigma}{\Gamma \vdash x{:}\sigma}$$

$$\text{(ABS)} \qquad \frac{\Gamma[x{:}\sigma] \vdash e{:}\sigma'}{\Gamma \vdash \lambda x.\, e{:}\sigma \to \sigma'}$$

$$\text{(APP)} \qquad \frac{\Gamma \vdash e{:}\sigma \to \sigma' \quad \Gamma \vdash e'{:}\sigma}{\Gamma \vdash ee'{:}\sigma'}$$

We call a λ-term E *well-typed* (or just *typed*) if $\Gamma \vdash E{:}\sigma$ is derivable by the above rules, for some Γ and σ.

In the above definition, we have adopted the "Curry View" of TLC, where types are inferred for unadorned terms using the (Var), (Abs), and (App) rules. Equivalently, we could have chosen the "Church View," where types and terms are defined together and λ-bound variables are annotated with their type (i.e., we would have $\lambda x{:}\sigma.\, e$ instead of $\lambda x.\, e$ in the (Abs) rule). In fact, *in our encodings below we will often provide type annotations to make the type of a term clear.*

For typed λ-terms e, e', we write $e \rhd_\alpha e'$ (α-reduction) when e' can be derived from e by renaming of a λ-bound variable, for example $\lambda x.\, \lambda y.\, y \rhd_\alpha \lambda x.\, \lambda z.\, z$. We write $e \rhd_\beta e'$ (β-reduction) when e' can be derived from e by replacing a subterm in e of the form $(\lambda x.\, E)E'$ by $E[E'/x]$ (E with E' substituted for all free occurrences of x in E). Reduction preserves types. Let $\rhd$ be the reflexive, transitive closure of $\rhd_\alpha$ and $\rhd_\beta$.

TLC$^=$: We obtain TLC$^=$ 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) introducing an equality constant Eq of type $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ (for some fixed type variable τ different from o). The type inference system is the same with one modification: the Γ's must treat the constants as free variables associated with the fixed types o and $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$, respectively.

The reduction rules of TLC$^=$ are obtained by enriching the operational semantics of TLC as follows. For every pair of constants $o_i, o_j{:}\mathsf{o}$, we add to $\rhd$ the reduction rule

$$(Eq\, o_i\, o_j) \rhd \begin{cases} \lambda x{:}\tau.\, \lambda y{:}\tau.\, x & \text{if } i = j, \\ \lambda x{:}\tau.\, \lambda y{:}\tau.\, y & \text{if } i \neq j. \end{cases}$$

These are known as delta reductions.

TLC and TLC$^=$ enjoy the following properties, see [15, 6]:

Church-Rosser: If $e \rhd e'$ and $e \rhd e''$, then there exists a λ-term e''' such that $e' \rhd e'''$ and $e'' \rhd e'''$.

Strong normalization: For each e, there exists an integer n such that if $e \rhd e'$, then the derivation involves no more than n individual β-reductions.

Principal Type: A typed λ-term E has a principal type, that is a type from which all other types can be obtained via substitution.

Type Inference: One can show that given E it is decidable in linear time whether E is a typed λ-term. Also, given $\Gamma \vdash E{:}\sigma$ it is decidable in linear time if this statement is derivable by the above rules. (Both these algorithms use first-order unification, e.g., see [26]. They work with or without type annotations and with or without constants in the Γ's.)

Functionality Order: 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 $o(\sigma' \to \sigma'') = \max(1 + o(\sigma'), o(\sigma''))$. We also refer to the order of a typed λ-term as the order of its type. Note that, the order of the fixed type variables o and τ is 0. The above definitions and properties hold for fragments of TLC and TLC$^=$, where order of terms is some fixed k. In such fragments we use the above inference rules (Var), (Abs), and (App), but with all types restricted to order k.

2.2 `let`-Polymorphism: Core-ML

Core-ML: The syntax of core-ML is the syntax of TLC augmented with one new expression construct: $\mathcal{E} \equiv x \mid (\mathcal{E}\mathcal{E}) \mid \lambda x.\mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ \mathcal{E}$. The simplest way of explaining ML types involves the same monomorphic types and rules (Var), (Abs), and (App) used for TLC with one additional rule that captures the polymorphism (see [26]):

$$\text{(LET)} \qquad \frac{\Gamma \vdash e'{:}\sigma' \quad \Gamma \vdash e[e'/x]{:}\sigma}{\Gamma \vdash \texttt{let}\ x = e'\ \texttt{in}\ e{:}\sigma}$$

We call a λ-term E *ML-typed* if $\Gamma \vdash E{:}\sigma$ is derivable by the (Var), (Abs), (App), and (Let) rules, for some Γ and σ. The operational semantics for $\texttt{let}\ x = M\ \texttt{in}\ N$ is the same as for $(\lambda x.\, N)M$. So core-ML has the same expressive power as TLC. However, core-ML allows more flexibility in typing.

For example, $\texttt{let}\ x = (\lambda z.\, z)\ \texttt{in}\ (xx)$ is in core-ML but $(\lambda x.\, xx)(\lambda z.\, z)$ is not in TLC; the equivalent

program in TLC is what we get after one reduction of $(\lambda x.\, xx)(\lambda z.\, z)$, namely $(\lambda z.\, z)(\lambda z.\, z)$.

The analogous definitions, expressibility, principal type and type inference properties hold for core-ML$^=$, where constants and their equality are added as in TLC$^=$. Order of functionality is defined in the same way. There are two differences: (1) Type inference is no longer in linear time but EXPTIME-complete [26, 27]. (2) Arbitrary order core-ML, core-ML$^=$, TLC, and TLC$^=$ all have the same expressive power, but for fixed order type inference allows more core-ML than TLC programs to be typed, so it might provide more expressibility.

2.3 Elementary Recursion via List Iteration

We briefly review how list iteration works. 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 when we compute). We abbreviate this list construction as $[x_1, x_2, \ldots, x_k]$; the variables c and n abstract over the constructors *cons* and *nil*.

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 $\mathsf{Bool} \equiv \tau \to \tau \to \tau$. Define the exclusive or as $\mathit{Xor} \equiv \lambda p{:}\,\mathsf{Bool}.\, \lambda q{:}\,\mathsf{Bool}.\, \lambda x{:}\,\tau.\, \lambda y{:}\,\tau.\, p(qyx)(qxy)$, and the parity of a list of Boolean values as

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

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. List iteration is a powerful programming technique, which can be used in the context of TLC and TLC$^=$ to encode any elementary recursion [37, 33]. However, some care is needed if one is to maintain well-typedness [21].

3 Representing Databases and Queries

3.1 Databases as Lambda Terms

Relations are represented in our framework 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.

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* $\overline{r}$ of r is the λ-term

$$\lambda c.\, \lambda n.\ (c\, o_{1,1}\, o_{1,2} \ldots o_{1,k}\ \ldots\ (c\, o_{2,1}\, o_{2,2} \ldots o_{2,k}\ \ldots\ (c\, o_{m,1}\, o_{m,2} \ldots o_{m,k}\, n)\ldots))\,,$$

which 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.

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.[1] 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, or, if the exact nature of θ does not matter, as o_k^*.

It is fairly easy to see that a principal type of o_k^σ characterizes the generalized Church numerals in the following sense:

Lemma 3.1 *Let f be any TLC$^=$ term without free variables and in normal form (i.e., with no beta- or delta-reduction possible) of type o_k^σ, where σ is a type variable different from o. Then either $f \equiv \lambda c.\, co_{1,1} \ldots o_{1,k}$ or $f \equiv \overline{r}$ for some relation $r \subseteq O^k$.*

Remark: Since the two terms $\lambda c.\, co_{1,1} \ldots o_{1,k}$ and $\lambda c.\, \lambda n.\, co_{1,1} \ldots o_{1,k} n$, η-convert (see [6]) to each other, they cannot be distinguished at the type level. For this reason, we allow both forms as valid representations of relations containing just one tuple.

3.2 Query Languages

We now define our query languages. For purposes of comparison we use the same syntax in both TLI and MLI definitions. That is, in MLI we interpret the outermost λ's as `let`'s. The other `let`'s in MLI can be eliminated, without problems of expressibility or (as we later show) type inference.

Definition 3.2 *A query program of arity $(k_1, \ldots, k_l, k)$ in* TLI$_i^=$ *(the language of* typed list iteration of order $i+3$ with equality*) is a typed TLC$^=$ term Q of order $i+3$ such that: Q has the form $\lambda R_1 \ldots \lambda R_l.\, M$ and for every database of arity $(k_1, \ldots, k_l)$ encoded by $\overline{r_1} \ldots \overline{r_l}$ it is possible to type $(\lambda R_1 \ldots \lambda R_l.\, M)\overline{r_1} \ldots \overline{r_l}$ as o_k^τ.*

Definition 3.3 *A query program of arity $(k_1, \ldots, k_l, k)$ in* MLI$_i^=$ *(the language of* ML-typed list iteration of order $i+3$ with equality*) is a typed core-ML$^=$ term Q of order $i+3$ such that: Q has the form $\lambda R_1 \ldots \lambda R_l.\, M$ and for every database of arity $(k_1, \ldots, k_l)$ encoded by $\overline{r_1} \ldots \overline{r_l}$ it is possible to ML-type $(\lambda R_1 \ldots \lambda R_l.\, M)\overline{r_1} \ldots \overline{r_l}$ as o_k^τ with the bindings $\lambda R_1 \ldots \lambda R_l$ typed as `let`'s.*

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

These definitions are semantic because they involve quantification over all inputs. By Lemma 3.1 and the fact that τ is a type variable different from o, it is easy to see that every program in these languages is guaranteed to have a correct output given correct inputs. These semantic definitions can be made syntactic:

Lemma 3.4 *Given $(k_1, \ldots, k_l, k)$ and a typed λ-term $(\lambda R_1 \ldots \lambda R_l.\, M)$ of TLC$^=$ or core-ML$^=$ of order $i+3$ one can efficiently decide if it is a query program of arity $(k_1, \ldots, k_l, k)$ of TLI$_i^=$ or MLI$_i^=$. Moreover, all inputs to this term can be typed with the same monomorphic type.*

Remark: In the setting of these query languages input and output terms are monomorphically typed. However, unlike [19, 36], we allow that the monomorphic types of inputs and outputs differ. Outputs are always typed as o_k^τ but inputs can be typed as o_k^*. This convention is necessary for expressing all of PTIME.

4 Embedding Queries in TLI$_0^=$ and TLI$_1^=$

To illustrate the power of list iteration, we show how to express various well-known database queries in TLI$_0^=$ and TLI$_1^=$. Our encodings show that TLI$_0^=$ expresses relational algebra and that TLI$_1^=$ expresses all PTIME queries. If, in addition to the typing $Eq\colon o \to o \to \tau \to \tau \to \tau$ prescribed in Section 2.1, we also allow Eq to be typed as $Eq\colon o \to o \to o \to o \to o$ (thereby introducing a weak form of polymorphism), we obtain a version TLI$_0^\approx$ of TLI$_0^=$ that expresses relational algebra, parity, majority, and (deterministic) graph accessibility.

4.1 Embeddings in TLI$_0^=$ and TLI$_0^\approx$

Relational Algebra: In [21], we showed how to express relational algebra using list iteration. Due to a different input/output format, our encodings involved λ-terms of order 5. With straightforward modifications, these terms work under the present input/output conventions and the rank drops down to 3. We give the Cartesian product and intersection operators as examples and refer the reader to [21] for the other operators.

$$
\begin{array}{l}
\mathit{Times}\colon o_k^\tau \to o_l^\tau \to o_{k+l}^\tau \equiv \\
\quad \lambda R\colon o_k^\tau.\, \lambda S\colon o_l^\tau. \\
\qquad \lambda c\colon o \to \cdots \to o \to \tau \to \tau.\, \lambda n\colon \tau. \\
\qquad\quad R\,(\lambda x_1\colon o \ldots \lambda x_k\colon o.\, \lambda T\colon \tau. \\
\qquad\qquad S\,(\lambda y_1\colon o \ldots \lambda y_l\colon o.\, \lambda U\colon \tau. \\
\qquad\qquad\quad c\, x_1 \ldots x_k\, y_1 \ldots y_l\, U)\, T)\, n
\end{array}
$$

$$
\begin{array}{l}
\mathit{Intersection}\colon o_k^\tau \to o_k^\tau \to o_k^\tau \equiv \\
\quad \lambda R\colon o_k^\tau.\, \lambda S\colon o_k^\tau. \\
\qquad \lambda c\colon o \to \cdots \to o \to \tau \to \tau.\, \lambda n\colon \tau. \\
\qquad\quad R\,(\lambda x_1\colon o \ldots \lambda x_k\colon o.\, \lambda T\colon \tau. \\
\qquad\qquad (\mathit{Member}\, x_1 \ldots x_k\, S)\, (c\, x_1 \ldots x_k\, T)\, T)\, n
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{Member}\colon \overbrace{o \to \cdots \to o}^{k} \to o_k^\tau \to \mathsf{Bool} \equiv \\
\quad \lambda x_1\colon o \ldots \lambda x_k\colon o.\, \lambda R\colon o_k^\tau. \\
\qquad \lambda u\colon \tau.\, \lambda v\colon \tau. \\
\qquad\quad R\,(\lambda y_1\colon o \ldots \lambda y_k\colon o.\, \lambda T\colon \tau. \\
\qquad\qquad Eq\, x_1\, y_1\, (Eq\, x_2\, y_2 \ldots (Eq\, x_k\, y_k\, u\, T)\, T) \ldots T)\, v
\end{array}
$$

Parity: The following term computes whether a relation R contains an odd or even number of tuples. If the cardinality of R is even, the output is the singleton list [1], otherwise it is the singleton list [0] (here 0 and 1 are TLC constants). The type of Eq in this example is $o \to o \to o \to o \to o$, i.e., this term is a TLI$_0^\approx$ query.

$$
\begin{array}{l}
\mathit{Parity}\colon o_k^\circ \to o_1^\tau \equiv \\
\quad \lambda R\colon o_k^\circ. \\
\qquad \lambda c\colon o \to \tau \to \tau.\, \lambda n\colon \tau. \\
\qquad\quad c\,(R\,(\lambda x_1\colon o \ldots \lambda x_k\colon o.\, \lambda P\colon o.\, (Eq\, P\, 0)\, 1\, 0)\, 0)\, n
\end{array}
$$

Majority: Here the input consists of a binary relation R, where each tuple contains a unique constant in the first column (to make the tuple unique) and either the constant 1 or the constant 0 in the second column. The task is to determine whether there are more 1's than 0's in the second column. The following term decides this, reducing to [1] if the answer is "yes" and to [0] otherwise. It uses the "labels" in the first column of R as *numbers*, treating the (unique) constant in the first column of the i-th tuple of R as the number $i-1$. Again, this term is a TLI$_0^\approx$ query.

$$
\begin{array}{l}
\mathit{Majority}\colon o_2^\circ \to o_1^\tau \equiv \\
\quad \lambda R\colon o_2^\circ. \\
\qquad \lambda c\colon o \to \tau \to \tau.\, \lambda n\colon \tau. \\
\qquad\quad c\,(\mathit{Compare}_R \\
\qquad\qquad (R\,(\lambda x_1\colon o.\, \lambda x_2\colon o.\, \lambda T\colon o. \\
\qquad\qquad\quad (Eq\, x_2\, 1)\, (\mathit{Succ}_R\, T)\, T)\, \mathit{First}_R) \\
\qquad\qquad (R\,(\lambda x_1\colon o.\, \lambda x_2\colon o.\, \lambda T\colon o. \\
\qquad\qquad\quad (Eq\, x_2\, 0)\, (\mathit{Succ}_R\, T)\, T)\, \mathit{First}_R) \\
\qquad\qquad 0\, 0\, 1)\, n
\end{array}
$$

Here, First_R, Succ_R, and $\mathit{Compare}_R$ are functions that operate on the "labels" in the first column of R. First_R returns the label of the first tuple in R, $(\mathit{Succ}_R\, x)$ returns the label of the tuple following the one labeled x, and $(\mathit{Compare}_R\, x\, y\, a\, b\, c)$ compares the positions in R of the tuples labeled x and y, reducing to a if x precedes y, to b if x and y are equal, and to c if y precedes x. These

terms can be written as follows:

$$First_R \equiv R\,(\lambda x_1\!:\!o.\,\lambda x_2\!:\!o.\,\lambda T\!:\!o.\,x_1)\,0$$

$$(Succ_R\,x) \equiv R\,(\lambda x_1\!:\!o.\,\lambda x_2\!:\!o.\,\lambda T\!:\!o.\,Compare_R\,x_1\,x\,T\,T\,x_1)\,x$$

$$(Compare_R\,x\,y\,a\,b\,c) \equiv R\,(\lambda x_1\!:\!o.\,\lambda x_2\!:\!o.\,\lambda T\!:\!o.\; Eq\,x\,y\,b\,(Eq\,x_1\,x\,a\,(Eq\,x_1\,y\,c\,T)))\,b$$

Deterministic Graph Accessibility: Suppose that G is a directed graph in which each node has at most one outgoing edge. The (deterministic) graph accessibility problem consists of determining, for two given vertices (u, v), whether there is a path in G from u to v. We assume that G is given as a binary relation R containing tuples of the form $(x, Parent(x))$ and that S is a binary relation containing a single tuple (u, v). The following $\mathrm{TLI}_0^{\approx}$ term decides whether u is an ancestor of v in G, reducing to [1] if the answer is "yes" and to [0] otherwise. The idea is to use the list R twice: in an inner loop, to compute the parent of a vertex, and in an outer loop, to iterate the parent operation until either the desired vertex is found or $|R|$ iterations have been done.

$$DGAP\!:\!o_2^o \to o_2^o \to o_1^\tau \equiv \lambda R\!:\!o_2^o.\,\lambda S\!:\!o_2^o.\; \lambda c\!:\!o \to \tau \to \tau.\,\lambda n\!:\!\tau.\; c\,(S\,(\lambda uv\!:\!o.\,\lambda W\!:\!o.\,Eq\,v\,(Ancestor\,u)\,1\,0)\,0)\,n\,,$$

where

$$(Ancestor\,u) \equiv R\,(\lambda x_1\!:\!o.\,\lambda x_2\!:\!o.\,\lambda T\!:\!o.\,(Eq\,T\,v)\,T\,(Parent\,T))\,u$$

and

$$(Parent\,v) \equiv R\,(\lambda x_1\!:\!o.\,\lambda x_2\!:\!o.\,\lambda T\!:\!o.\,(Eq\,x_1\,v)\,x_2\,T)\,v$$

It is interesting to note that deterministic graph accessibility is LOGSPACE-complete for first-order reductions [24], but only if vertices can be labeled by *tuples of constants*. This means that an instance of the problem consists of a $2k$-ary relation R such that each tuple $(x_1, \ldots, x_k, y_1, \ldots, y_k) \in R$ denotes an edge from the vertex labeled $(x_1, \ldots, x_k)$ to the vertex labeled $(y_1, \ldots, y_k)$.

It seems that this more general version of graph accessibility cannot be expressed in $\mathrm{TLI}_0^{\approx}$, since it requires list iteration over *tuples* of constants, which cannot be encoded as $\mathrm{TLC}^=$ objects of order 0. Thus, the expressive power of $\mathrm{TLI}_0^{\approx}$ appears to fall short of LOGSPACE. It is possible to express all of LOGSPACE by adding tuples of constants as primitive objects to the language, but this would sacrifice the simplicity of the framework to some extent. (It can be shown that the $\mathrm{TLI}_0^{=} \subseteq$ LOGSPACE result of Section 6 holds true even if $\mathrm{TLI}_0^{=}$ is augmented with a polymorphic equality and tuples; so $\mathrm{TLI}_0^{\approx}$ + tuples = LOGSPACE.)

4.2 Embeddings in $\mathrm{TLI}_1^{=}$

$\mathrm{TLI}_0^{=}$ is not powerful enough to compute fixpoints of relational queries, because the language only allows the iteration of mappings from order-zero objects to order-zero objects. It is necessary to go to $\mathrm{TLI}_1^{=}$ so as to iterate mappings from relations to relations. That $\mathrm{TLI}_1^{=}$ is sufficient follows from the encodings given in [21], plus the fact that over a known domain, relations can be represented by order-one objects, namely *characteristic functions*. The characteristic function f_r of a k-ary relation r is a $\mathrm{TLC}^=$ term of type $o \to \cdots \to o \to \mathsf{Bool}$, such that for any k constants $o_{i_1}, \ldots, o_{i_k}$,

$$(f_r\,o_{i_1} \ldots o_{i_k}\,u\,v) \rhd \begin{cases} u & \text{if } (o_{i_1}, \ldots, o_{i_k}) \in r, \\ v & \text{if } (o_{i_1}, \ldots, o_{i_k}) \notin r. \end{cases}$$

Since the domain of a query can be computed from the input relations (by forming the union of all columns), it is possible to write λ-terms *FuncToList* and *ListToFunc* that translate between the iterator and characteristic function representation of a relation. Using these operators, a fixpoint query can be expressed in $\mathrm{TLI}_1^{=}$ essentially as follows:

$$\Upsilon \equiv \lambda R_1 \ldots \lambda R_l.\; FuncToList\;(Crank\;(\lambda \vec{x}.\,\lambda f.\,ListToFunc\,(Q\,(FuncToList\,f)))\;(ListToFunc\,Nil))\,,$$

where $Q = \lambda R.\,Q'$ is the encoding of the first-order query to be iterated (with $R_1, \ldots, R_l$ occurring free in Q), $Nil = \lambda c.\,\lambda n.\,n$ denotes the empty list, and *Crank* is a sufficiently large cross product of the input relations, serving as a "crank" to iterate Q a polynomial number of times.

As explained in [21], additional care is necessary to make Υ typable using monomorphic types. This is because the inputs $R_1, \ldots, R_l$ are used to iterate both over order-one objects (in *Crank*) and over order-zero objects (in Q). With monomorphic types, this is normally impossible. However, [21] shows how to get around this problem by introducing a "type-laundering" operator that essentially turns iterations over order-zero objects into iterations over order-one objects. By using this operator inside Q, the term Υ becomes typable in the monomorphic type system.

A much simpler way around this problem is the use of `let`-polymorphism: By rewriting Υ as

$$\texttt{let}\ R_1 = \overline{r_1}\ \texttt{in} \ldots \texttt{let}\ R_l = \overline{r_l}\ \texttt{in}$$

$$
\begin{array}{l}
\mathit{FuncToList} \\
\quad (\mathit{Crank} \\
\quad\quad (\lambda \vec{x}.\, \lambda f.\, \mathit{ListToFunc}\,(Q\,(\mathit{FuncToList}\, f))) \\
\quad\quad (\mathit{ListToFunc}\, \mathit{Nil}))\,,
\end{array}
$$

where $\overline{r_1}, \ldots, \overline{r_l}$ are the encodings of the input relations, the variables $R_1, \ldots, R_l$ are declared to be polymorphic, so it does not matter that their occurrences in *Crank* and Q require different types. We will show in the next two sections that the presence of `let` does not affect the expressive power of $\mathrm{TLI}_1^=$ and that for fixed order, `let`-expressions can be type checked in polynomial time, so the introduction of `let`-polymorphism facilitates a more natural programming style at no additional cost.

5 `let`-Polymorphism and $\mathrm{MLI}_1^=$

As shown in the previous section, ML polymorphism provides flexibility in programming fixpoints. The `let` construct is used in the various MLI's to receive the inputs, but also can be used in the body of the program. The occurrences of `let` in the program body can be eliminated by reduction at the expense of program body length [26]. A problem with use of `let` in the program body is that type inference may become inefficient. We show, however, that the fixed order restriction can be used to eliminate this inefficiency.

Theorem 5.1 *For each fixed k, type inference in order k core-ML$^=$ is polynomial in the program size.*

The proof has two parts. The first part involves the rules (Var), (Abs), and (App). In general, to achieve PTIME type inference in TLC one must use directed acyclic graph representations of types. For fixed order, we show that tree representations of unbounded fan-out and fixed depth suffice. The second part involves the rule (Let). Using the tree representations of the first part it is possible to produce a polynomial bound on the fan-out of the tree representations.

6 PTIME and the Power of $\mathrm{TLI}_1^=$

In this section, we show that $\mathrm{TLI}_1^=$ and $\mathrm{MLI}_1^=$ queries can be evaluated in time polynomial in the size of the input relations. The evaluation algorithm is essentially a λ-reduction engine, augmented with certain "optimizations" made possible by the restrictions on the I/O-behavior and the order of the query term. These "optimizations" ensure that all terms occurring during the reduction sequence are of polynomial size.

6.1 The Structure of $\mathrm{TLI}_1^=$ Terms

In the following, let Q be a fixed $\mathrm{TLI}_1^=$ or $\mathrm{MLI}_1^=$ term. We can assume that Q is in normal form, because the reduction to normal form can be done in a preprocessing step that does not figure in the data complexity of the query. We can also eliminate all `let`-expressions from Q by replacing every subterm of the form "`let` $x = N$ `in` M" with $M[N/x]$ and by agreeing that variables corresponding to input relations are to be polymorphically typed.

It is convenient to introduce some terminology for the subterms of Q. Since Q is in normal form, every subterm of Q is of the form $\lambda x_1.\, \lambda x_2 \ldots \lambda x_k.\, f M_1 \ldots M_l$, where $k, l \geq 0$, $x_1, \ldots, x_k$ and f are variables, and $M_1, \ldots, M_l$ are terms. An occurrence of a subterm T is called *complete* if k and l are maximal, i.e., if the occurrence is not of the form $(\lambda x.\, T)$ or (TS). In this case, $M_1, \ldots, M_l$ are called the *arguments* of f and f is called the function symbol *governing* the occurrence of M_i for $1 \leq i \leq l$. It is easy to see that for every occurrence of a subterm of Q, there is a smallest complete subterm containing that occurrence. In particular, every occurrence of a variable in Q not immediately to the right of a λ is the governing symbol for a well-defined (but possibly empty) set of arguments.

In order to simplify the evaluation algorithm, we will first preprocess Q into an equivalent query term with certain structural properties. This transformation is independent of any input relations, i.e., its data complexity is $O(1)$. The following definition specifies the special kind of term the evaluation algorithm operates on.

Definition 6.1 *Let Q be a $\mathrm{TLI}_1^=$ term mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Q is said to be in* canonical form *if the following conditions are true:*

1. *Q is of the form $\lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$.*
2. *Every occurrence of R_i in Q' (where $1 \leq i \leq l$) is of the form $R_i(\lambda x_1 \ldots \lambda x_{k_i}.\, \lambda f.\, M) N T_1 \ldots T_m$, where k_i is the arity of the i-th input relation, f is a variable of order ≤ 1, M and N are terms of order ≤ 1, and $T_1, \ldots, T_m$ (where $m \geq 0$) are terms of order 0. We call f the* accumulator variable *for this occurrence of R_i.*
3. *Every occurrence of Eq in Q' has exactly 4 arguments.*
4. *Every occurrence of c in Q' has exactly $k+1$ arguments, where k is the arity of the output relation.*
5. *Every occurrence of n in Q' has exactly 0 arguments.*
6. *The only (free or bound) variables in Q of non-zero order are $R_1, \ldots, R_l$, c, and accumulator variables.*
7. *Q is in normal form.*

Lemma 6.2 *Let P be a $\mathrm{TLI}_1^=$ term mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then there is a $\mathrm{TLI}_1^=$ term Q in canonical form such that P and Q define the same database query, i.e., for every legal input $\overline{r_1}, \ldots, \overline{r_l}$, the normal forms of $(P\overline{R_1}, \ldots, \overline{R_l})$ and $(Q\overline{R_1}, \ldots, \overline{R_l})$ encode the same relation. Q can be effectively determined from P.*

The proof involves executing a series of transformations of P that successively establish properties (1) to (7) of the canonical form without changing the semantics of P. For example, property (1) can be established by replacing P with $\lambda R_1.\,\lambda R_2 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,P R_1 R_2 \ldots R_l c n$ and property (2) can be established by replacing every occurrence of R_i in P by $(\lambda c.\,\lambda n.\,R_i(\lambda x_1 \ldots \lambda x_{k_i}.\,\lambda f.\,c x_1 \ldots x_{k_i} f) n)$ and reducing to normal form.

Lemma 6.3 *If $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ is in canonical form, then every complete subterm t of Q' has the form $\lambda \vec{x}.\,M$, where $\vec{x}$ is a vector of order-zero variables and possibly one accumulator variable and M has one of the following forms:*

1. $R_i(\lambda x_1{:}\mathsf{o} \ldots \lambda x_{k_i}{:}\mathsf{o}.\,\lambda f{:}\sigma.\,M) N T_1 \ldots T_m$,
2. $Eq\,S\,T\,U\,V$,
3. $c T_1 \ldots T_k T_{k+1}$,
4. $f T_1 \ldots T_m$, *where f is an accumulator variable,*
5. x, *where x is a variable of order 0 or a constant.*

(By definition of an accumulator variable, $\vec{x}$ contains an accumulator variable if and only if t is the first argument of an occurrence of some R_i.)

Proof: By property (6) of the canonical form, the only variables that may be λ-bound inside Q' are accumulator variables and variables of order 0, so only these can occur in $\vec{x}$.

Let s be the top-level symbol of M, i.e., $M = s M_1 \ldots M_n$. By property (6), there are five possibilities for s: it can be one of $R_1, \ldots, R_l$, in which case property (2) implies that M is of form (1); it can be Eq, in which case property (3) implies form (2); it can be c, in which case property (4) implies form (3); it can be an accumulator variable, in which case form (4) applies; or it can be a variable of order 0 or a constant, in which case form (5) applies. □

6.2 The Evaluation Algorithm

The formal specification of the evaluation algorithm is too long to be included here. Instead, we will give an informal description of the underlying ideas.

An evaluation algorithm for $\mathrm{TLI}_1^=$ terms essentially has to deal with the five kinds of expressions listed in Lemma 6.3. Once $R_1, \ldots, R_l$ are instantiated, these expressions normalize to terms of the form $\lambda \vec{x}.\,M$, where $\vec{x}$ is a (possibly empty) vector of order 0 variables and M is a λ-free term of order 0 built from Eq, c, variables of order 0, and constants. Unfortunately, these normal forms can be of exponential size for two reasons: (1) an exponential number of occurrences of Eq or (2) an exponential number of occurrences of c. A PTIME evaluation algorithm must deal with these two situations.

Problem (1) can be handled by the following observation. Even though a normal form t may contain an exponential number of occurrences of Eq, there is only a polynomial (in the size of the domain) number of different assignments of constants to variables of type o in t, thus many occurrences of Eq in t must be redundant. Suppose that $O = \{o_1, \ldots, o_N\}$ is the database universe and that t is of the form $\lambda \vec{x}.\,M$, where M is λ-free. Let $x_1, \ldots, x_m$ be the variables of M of type o and let $M_{i_1,\ldots,i_m}$ denote the normal form of $M[x_1{:=}\,o_{i_1}, \ldots, x_m{:=}\,o_{i_m}]$. Clearly, $M_{i_1,\ldots,i_m}$ does not contain any occurrences of Eq. Now consider the term

$$
\begin{aligned}
M' \equiv (&Eq\,x_1\,o_1 \\
&(Eq\,x_2\,o_1 \\
&\quad \ddots \\
&\quad (Eq\,x_m\,o_1\,M_{1,1,\ldots,1,1} \\
&\quad (Eq\,x_m\,o_2\,M_{1,1,\ldots,1,2} \\
&\quad\quad \vdots \\
&\quad (Eq\,x_m\,o_{N-1}\,M_{1,1,\ \ldots,1,N-1}\,M_{1,\ldots,1,N}))\ldots) \\
&\quad\quad \vdots \\
&(Eq\,x_2\,o_2 \\
&\quad \ddots \\
&\quad (Eq\,x_m\,o_1\,M_{1,2,\ldots,1,1} \\
&\quad (Eq\,x_m\,o_2\,M_{1,2,\ldots,1,2} \\
&\quad\quad \vdots
\end{aligned}
$$

This term has a polynomial number of occurrences of Eq arranged as a "decision tree" with the terms $M_{i_1,\ldots,i_m}$ at its leaves. Furthermore, M' is equivalent to M in the sense that for every choice of constants $(o_{i_1}, \ldots, o_{i_m})$, the terms $M[x_1{:=}\,o_{i_1}, \ldots, x_m{:=}\,o_{i_m}]$ and $M'[x_1{:=}\,o_{i_1}, \ldots, x_m{:=}\,o_{i_m}]$ convert to each other. Since the variables $x_1, \ldots, x_m$ must eventually be instantiated with constants anyway (the final output of a query does not contain any variables of type o), the evaluation algorithm can return the term $t' \equiv \lambda \vec{x}.\,M'$ instead of t without affecting the final result.

Problem (2) can be handled as follows. The terms $M_{i_1,\ldots,i_m}$ defined above are λ-free and contain only constants, variables of type τ, and the symbol c. It is easy to see that each such term must be either a constant, a variable, or a list-like structure

$$
\begin{aligned}
&c\,o_{1,1}\,o_{1,2} \ldots o_{1,k} \\
&\quad (c\,o_{2,1}\,o_{2,2} \ldots o_{2,k} \\
&\qquad \cdots \\
&\qquad\quad (c\,o_{m,1}\,o_{m,2} \ldots o_{m,k}\,x)) \ldots),
\end{aligned}
$$

where x is some variable of type τ. If such a term is of exponential size, then only because the list contains many duplicates. It is easy to see that elimination of these duplicates does not affect the

output relation produced by a query, even though it may cause the computed representation of the output to be different (it will be the duplicate-free version of the original representation). Thus, the evaluation algorithm is free to remove duplicates from every term $M_{i_1,\ldots,i_m}$ it constructs, thereby always returning terms of polynomial size.

Using the above polynomial-size representation of order 1 terms, the evaluation of a canonical form 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'. Subterms of the form $R_i(\lambda \vec{x}.\, \lambda f.\, M)NT_1 \ldots T_m$ are evaluated by evaluating N first and then evaluating the "loop body" M once for each tuple in r_i, from last to first, with $\vec{x}$ bound to the current tuple and f bound to the result of the previous iteration (in decision tree format). The final result of the loop is then applied to the evaluated values of $T_1 \ldots T_m$.

Subterms of the form $(Eq\, STUV)$ or $(cT_1 \ldots T_{k+1})$ are evaluated by evaluating the arguments first and then constructing a decision tree for the result. Finally, subterms of the form $fT_1 \ldots T_m$ are evaluated by substituting the evaluated arguments (which must be order 0 terms) into the decision tree for f.

It is easy to see that this procedure terminates after a number of steps polynomial in the size of $\overline{r_1}, \ldots, \overline{r_l}$ and that the work performed at each steps is polynomial as well. Also, it does not matter to the evaluation algorithm whether $R_1, \ldots, R_l$ are monomorphically or polymorphically typed. Hence, we have the following result:

Theorem 6.4 *Database queries defined by terms in $TLI_1^=$ and $MLI_1^=$ can be evaluated in PTIME.*

Combining this result with the encoding of fixpoint queries in $\mathrm{TLI}_1^=$ presented in Section 4.2, we obtain:

Theorem 6.5 *The database queries definable by $TLI_1^=$ and $MLI_1^=$ terms are exactly the PTIME queries.*

Note that TLI and MLI queries can discern the ordering of the tuples in the input encoding (see, e.g., the *Compare* operator in Section 4.1), so $\mathrm{TLI}_1^=$ and $\mathrm{MLI}_1^=$ express *all* PTIME queries, not just the generic ones.

If the evaluation strategy described above is specialized to $\mathrm{TLI}_0^=$ and $\mathrm{MLI}_0^=$ terms, it can be shown that the resulting algorithm can be performed in logarithmic space. Thus, we have:

Theorem 6.6 *Database queries defined by terms in $TLI_0^=$ and $MLI_0^=$ can be evaluated in LOGSPACE.*

7 Eliminating Equality: PTIME in TLI_2

Once list iteration over order 2 objects is allowed, it becomes possible to express PTIME queries in the "pure" calculus, i.e., without Eq and constants. This is done by coding the constants as projection functions (of order 1) and writing a λ-term Eq (of order 2) that tests two projection functions for equality. More precisely, if the database universe is the set $O = \{o_1, \ldots, o_N\}$, then the i-th atom is encoded as the projection function

$$\pi_i^N \equiv \lambda x_1 \ldots \lambda x_N.\, x_i$$

and equality is encoded as

$$\lambda p.\, \lambda q.\, \lambda u.\, \lambda v.\, p\,(q\,u\overbrace{v \ldots v}^{N-1})\,(q\,v\,u\overbrace{v \ldots v}^{N-2}) \ldots (q\overbrace{v \ldots v}^{N-1}u)$$

which, when applied to two projection functions π_i^N and π_j^N, reduces to $\lambda uv.\, u$ if $i = j$ and to $\lambda uv.\, v$ otherwise.

Relations are encoded as iterators in the usual way, except that explicit constants are replaced by the corresponding projection functions. Note that the arity of the projection functions changes with the size of the database universe, so the encoding of a relation r depends not only on r itself, but also on the database that r appears in. The same goes for the equality predicate: different databases may need different encodings of Eq. Hence, *in this setting Eq has to be part of the input.*

It is easy to see that the encoding of fixpoint queries described in Section 4.2 works unchanged in this new setting, except that the symbol Eq has to be λ-bound at the outermost level. The order of the query terms increases by 1, because the characteristic function of a relation now becomes an order 2 object (mapping k projection functions to a Boolean). It follows that:

Theorem 7.1 *TLI_2 expresses every PTIME query.*

8 Conclusions and Open Problems

We have presented embeddings of database query languages in low order fragments of the typed λ-calculus as well as a new functional characterization of PTIME. We have shown that in fixed order fragments of the typed λ-calculus there is sufficient expressive power for the PTIME-queries and that type inference is efficient.

A number of interesting open problems remain. (1) Determine the exact expressive power of TLI and MLI for $i = 0$ and various versions of equality. (2) Determine the expressive power for TLI_2, as well as for higher orders, see [28, 22, 30]. (3) Determine functional characterizations of other complexity classes, in particular NP, PHIER and PSPACE, see [18, 23, 38, 5]. (4) Study optimal reduction strategies [32] in the TLC. (5) Study languages that combine list iterators and set iterators ala [8, 10, 9, 25, 39].

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, C. Beeri, M. Gyssens, and D. Van Gucht. An Introduction to the Completeness of Languages for Complex Objects and Nested Relations. In S. Abiteboul, P. Fischer, and H. Schek, editors, *Nested Relations and Complex Objects in Databases*, pp. 117–138. LNCS 361, Springer Verlag, 1987.

[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 and V. Vianu. Datalog Extensions for Database Queries and Updates. *J. Comput. System Sci.*, **43** (1991), pp. 62–124.

[5] S. Abiteboul and V. Vianu. Generic Computation and its Complexity. In *Proceedings of the 23rd ACM STOC* (1991), pp. 209–219.

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

[7] S. Bellantoni and S. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. In *Proceedings of the 24th ACM STOC* (1992), pp. 283–293.

[8] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings DBPL3*, pp. 9–19. Morgan-Kaufmann, 1991.

[9] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *Proceedings 4th ICDT*, pp. 140–154. LNCS 646, Springer Verlag, 1992.

[10] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *Proceedings of the 18th ICALP*, pp. 60–75. LNCS 510, Springer Verlag, 1991.

[11] P. Buneman, R. Frankel, and R. Nikhil. An Implementation Technique for DB Query Languages. *ACM Trans. on Database Systems*, **7** (1982), pp. 164–186.

[12] A. Chandra and D. Harel. Computable Queries for Relational Databases. *J. Comput. System Sci.*, **21** (1980), pp. 156–178.

[13] A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *J. Comput. System Sci.*, **25** (1982), pp. 99–128.

[14] A. Chandra and D. Harel. Horn Clause Queries and Generalizations. *J. Logic Programming*, **2** (1985), pp. 1–15.

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

[16] A. Cobham. The Intrinsic Computational Difficulty of Functions. In Y. Bar-Hillel, editor, *International Conference on Logic, Methodology, and Philosophy of Science*, pp. 24–30. North Holland, 1964.

[17] E. Codd. Relational Completeness of Database Sublanguages. In R. Rustin, editor, *Database Systems*, pp. 65–98. Prentice Hall, 1972.

[18] R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. *SIAM-AMS Proceedings*, **7** (1974), pp. 43–73.

[19] S. Fortune, D. Leivant, and M. O'Donnell. The Expressiveness of Simple and Second-Order Type Structures. *J. of the ACM*, **30** (1983), pp. 151–185.

[20] Y. Gurevich. Algebras of Feasible Functions. In *Proc. of the 24th IEEE FOCS* (1983), pp. 210–214.

[21] G. Hillebrand, P. Kanellakis, and H. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. In *Proceedings of the 8th IEEE LICS* (1993), pp. 332–343.

[22] R. Hull and J. Su. On the Expressive Power of Database Queries with Intermediate Types. *J. Comput. System Sci.*, **43** (1991), pp. 219–267.

[23] N. Immerman. Relational Queries Computable in PTIME. *Info. and Comp.*, **68** (1986), pp. 86–104.

[24] N. Immerman. Languages that Capture Complexity Classes. *SIAM J. Comp.*, **68** (1986), pp. 86–104.

[25] N. Immerman, S. Patnaik, and D. Stemple. The Expressiveness of a Family of Finite Set Languages. In *Proceedings of the 10th ACM PODS* (1991), pp. 37–52.

[26] P. Kanellakis, H. Mairson, and J. Mitchell. Unification and ML-type Reconstruction. In *Computational Logic: Essays in Honor of Alan Robinson*, pp. 444–478. MIT Press, 1991.

[27] A. Kfoury, J. Tiuryn, and P. Urzyczyn. An Analysis of ML Typability. In *Proceedings 17th Colloquium on Trees, Algebra and Programming*, pp. 206–220. LNCS 431, Springer Verlag, 1990.

[28] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The Hierarchy of Finitely Typed Functional Programs. In *Proceedings 2nd IEEE LICS* (1987), pp. 225–235.

[29] P. Kolaitis and C. Papadimitriou. Why Not Negation By Fixpoint? *J. Comput. System Sci.*, **43** (1991), pp. 125–144.

[30] G. Kuper and M. Vardi. On the Complexity of Queries in the Logical Data Model. *Theoretical Comput. Sci.*, **116** (1993), pp. 33–57.

[31] D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. In *Proc. of the Inter. Conf. on Typed Lambda Calculi and Applications*, Utrecht 1993. (To appear in *Fundamenta Informaticae*.)

[32] J.-J. Lévy. Optimal Reductions in the Lambda-Calculus. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pp. 159–191. Academic Press, 1980.

[33] H. Mairson. A Simple Proof of a Theorem of Statman. *TCS*, **103** (1992), pp. 387–394.

[34] R. Milner. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.*, **17** (1978), pp. 348–375.

[35] R. Milner. The Standard ML Core Language. *Polymorphism*, **2** (1985), 28 pp.

[36] H. Schwichtenberg. Definierbare Funktionen im λ-Kalkül mit Typen. *Archiv für mathematische Logik und Grundlagenforschung*, **17** (1976), pp. 113–114.

[37] R. Statman. The Typed λ-Calculus is not Elementary Recursive. *Theoretical Computer Sci.*, **9** (1979), pp. 73–81.

[38] M.Y. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM STOC* (1982), pp. 137–146.

[39] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings 12th ACM PODS* (1993), pp. 26–36.