Functional Database Query Languages as Typed Lambda Calculi of Fixed Order

Gerd G. Hillebrand and Paris C. Kanellakis

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-26
May 1994

Functional Database Query Languages as Typed Lambda Calculi of Fixed Order *

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

Paris C. Kanellakis‡
Brown University
pck@cs.brown.edu

Abstract

We present functional database query languages expressing the FO- and PTIME-queries. This framework is a functional analogue of the logical languages of first-order and fixpoint formulas over finite structures, and its formulas consist of: atomic constants of order 0, equality among these constants, variables, application, lambda and `let` abstraction; all typable in ≤ 4 functionality order. In this framework, proposed in [25] for arbitrary functionality order, typed lambda terms are used for input-output databases and for query program syntax, and reduction is used for query program semantics. 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 input-output databases requires at least order 3. We show that, over list-represented databases, both $\mathrm{TLI}_0^=$ and $\mathrm{MLI}_0^=$ exactly express the FO-queries and both $\mathrm{TLI}_1^=$ and $\mathrm{MLI}_1^=$ exactly express the PTIME-queries, where list-represented means that each relation is given as a list of tuples instead of a set. We also study type reconstruction when, as in the above languages, functionality order is bounded by a constant. We show that ML-type reconstruction is NP-hard in the size of the program typed, for each $\mathrm{MLI}_i^=$ with $i \geq 1$. This complements the EXPTIME-hardness results of [31, 32], which require programs of unbounded functionality order[1]. Thus, the common practice of programming with low order functionalities implies no loss of expressibility for feasible queries, but does not avoid the worst-case intricacies of ML-type reconstruction.

1 Introduction

Database Query Languages: The logical framework of first-order (FO) and fixpoint formulas over finite structures has been the principal vehicle of theoretical research in database query languages; see [11, 12, 16, 17] 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* or *algebra* [16], *Datalog*$^\neg$ and various *fixpoint logics* [3, 4, 12, 13, 34]. Most importantly, as shown in [28, 46], every PTIME-query can be expressed using Datalog$^\neg$ on ordered structures; and, as shown in [3], it suffices to use Datalog$^\neg$ syntax under a variety of semantics to express various fixpoint logics. We refer to [2] for a complete exposition of the logical framework and for the definitions of *FO-* and *PTIME-queries*.

*A preliminary summary of the work presented here appeared in [26].

†Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225, and by an ERCIM fellowship.
‡Research supported by ONR Contracts N00014-94-1-1153 and N00014-91-J-4052, ARPA Order 8225.

[1] It also corrects an error in an earlier version of this paper [26], where we claimed PTIME membership.

Extensions of this logical framework, based on high-order formulas over finite structures, have been proposed in order to manipulate complex-object databases e.g., [1]. Despite some success of these extensions, the resulting query languages do not capture all the features of object-oriented database (oodb) languages, a research area of much current practical interest. Functional programming, with its emphasis on abstraction and on data types, might provide more insight into oodb query languages; and is the paradigm of choice in many oodbs. Thus, there is a growing body of work on functional query languages, from the early FQL language of [10] to the more recent work on structural recursion as a query language for complex-objects [7, 8, 9, 30, 47].

In this context, it is natural to ask: "Is there a functional analogue of the logical framework of first-order and fixpoint formulas over finite structures?" In [25] 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 (functional) order is a measure of the nesting of type functionalities. We show that these fragments are functional analogues of the relational calculus/algebra and of fixpoint characterizations of PTIME.

TLC Expressibility and Finite Model Theory: The *simply typed λ-calculus* [14] (*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. In our definitions, we use the "Curry style" of TLC terms without type annotations and reconstruct types. For clarity of exposition we often provide the annotations in "Church style".

The expressive power of TLC was originally analyzed in terms of computations on simply typed Church numerals (see, e.g., [5, 18, 42]). 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 hard decision problems can be embedded in TLC, see [38, 39, 43], and that different typings allow exponentiation [18]. However, very few connections have been established between complexity theory and the λ-calculus.

One such connection was recently demonstrated by Leivant and Marion [36], who express all of PTIME while avoiding the anomalies associated with representations over Church numerals. In [36], the simply typed lambda calculus is augmented 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. With this addition, Leivant and Marion obtain various calculi in which there exist simple characterizations of PTIME. (Note that, since Cobham's early work there have been a number of interesting functional characterizations of PTIME, e.g., [6, 15, 20, 22], which are not directly related to the TLC).

It seems that to exhibit the power of the TLC one must add features as in [36] and/or modify the input-output conventions. Thus, in [25] we examine the expressive power of the typed λ-calculus, but over appropriately typed encodings of input-output finite structures. We also use TLC$^=$, the typed λ-calculus with atomic constants and an equality on them, and the associated delta-reduction of [14]. In [25], we examine both the "pure" TLC and the "impure" TLC$^=$ and show that: (a) TLC (or TLC$^=$) expresses exactly the elementary queries and, thus, is a functional language for the complex-object queries of [1]. (b) Every PTIME-query can be PTIME-embedded in TLC (or TLC$^=$), i.e., its evaluation can be performed in polynomial time with a simple reduction strategy. (c) Similar PTIME-embeddings exist for every FO-query using terms of order at most 3 in TLC$^=$ or order at most 4 in TLC. (d) Every PTIME-query can be embedded in TLC$^=$ using terms of order at most 4 or in TLC using terms of order at most 5.

The embeddings in (c-d) above, which minimized functionality order, were the starting point

for the work reported here. In particular, the embeddings in (d) presented some problems. Even if they expressed PTIME-queries, most reduction strategies required an exponential number of steps for evaluation. Unlike the reduction strategies of [25], the strategies presented here make critical use of auxiliary data structures to force a PTIME number of steps.

Contributions: In this paper we analyze fixed order fragments of $\text{TLC}^=$ and core-$\text{ML}^=$. By core-$\text{ML}^=$ we mean $\text{TLC}^=$ with `let`-polymorphism. This is the core part of Milner's ML language [23, 40], which combines the convenience of type reconstruction and the flexibility of polymorphism.

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 4 functionalities. In this framework, relations are encoded as typed λ-terms denoting lists of tuples and queries are typed λ-terms that, when applied to encodings of input relations, reduce to an encoding of the output relation. We define two families of languages: $\text{TLI}_i^=$ or simply-typed list iteration of order $i+3$ with equality, and $\text{MLI}_i^=$ or ML-typed list iteration of order $i+3$ with equality (we use $i+3$ since our list representation of input-output databases requires at least order 3). We study the *$\text{TLI}_i^=$-queries* and *$\text{MLI}_i^=$-queries*, that is the queries expressible in these languages over list-represented databases. *List-represented* means that each relation is given as an ordered list of tuples instead of a set. Our results are as follows:

1. We show: $\text{TLI}_0^=$-queries $\subseteq$ $\text{MLI}_0^=$-queries $\subseteq$ FO-queries, over list-represented databases. With the embedding of FO-queries into $\text{TLC}^=$ given in [25], this implies that $\text{TLI}_0^=$-queries = $\text{MLI}_0^=$-queries = FO-queries.

2. We show: $\text{TLI}_1^=$-queries $\subseteq$ $\text{MLI}_1^=$-queries $\subseteq$ PTIME-queries, over list-represented databases. With the embedding of PTIME-queries into $\text{TLC}^=$ given in [25], this implies that $\text{TLI}_1^=$-queries = $\text{MLI}_1^=$-queries = FO-queries. One consequence of this analysis is a functional characterization of PTIME that differs from those of [6, 15, 20, 22, 36] in the sense of having the fewest additions to TLC—just equality over atomic constants.

3. We derive an NP-hardness lower bound for the complexity of type reconstruction in fixed-order fragments of ML, such as $\text{MLI}_1^=$. This complements the EXPTIME-completeness results in [31, 32], which use terms of unbounded order.

Overview: The paper is organized as follows. We begin with a review of the simply typed λ-calculus in Section 2. To make our exposition self-contained we include all the necessary background in TLC, $\text{TLC}^=$ (Section 2.1), core-ML, core-$\text{ML}^=$ (Section 2.2), and list iteration (Section 2.3).

In Section 3, we show how to represent relational databases as λ-terms and we define families $\text{TLI}_i^=$, $\text{MLI}_i^=$ of query languages acting on such representations. In Section 4, we establish lower bounds on their expressive power. Since the techniques here are variants of those in [25], we only sketch the proofs. (For completeness we provide all related lambda terms in an Appendix).

In Section 5, we present the main analytic results of this paper. These are upper bounds on the expressibility of the $\text{TLI}_i^=$ and $\text{MLI}_i^=$ languages for $i = 0, 1$. To show these upper bounds we have to reason based on our input-output conventions. These proofs involve an analysis of the structure of programs and an evaluator of programs, which uses reduction plus specialized data structures.

In Section 6 we investigate the complexity of ML type reconstruction for terms of fixed order and derive an NP-hardness bound. The proof is a modification of the one given in [31]. It is based on the construction of terms with low functionality order, but high arity. We close with open questions in Section 7.

2 Programming in the Typed Lambda Calculus

2.1 The Simply Typed λ-Calculus: TLC and TLC$^=$

TLC: The syntax of TLC *types* is given by the grammar $\mathcal{T} := t \mid (\mathcal{T} \to \mathcal{T})$, where t ranges over a set of *type variables*. For example, α is a type, as are $(\alpha \to \beta)$ and $(\alpha \to (\alpha \to \alpha))$. We omit outermost parentheses and $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$.

The syntax of TLC λ-*terms* or *terms* is given by the grammar $\mathcal{E} := x \mid (\mathcal{E}\mathcal{E}) \mid \lambda x.\mathcal{E}$, where x ranges over a set of *term variables* (distinct from type variables) and where terms satisfy *typedness* as outlined below. As usual, a term t' is a *subterm* of another term t if t' occurs as part of t. We omit outermost parentheses and PQR stands for $(PQ)R$.

Typedness of terms is defined by the following inference rules, where Γ is a function from term variables to types, and $\Gamma \cup \{x : \tau'\}$ is the function Γ' updating Γ with $\Gamma'(x) = \tau'$:

(VAR) $$\frac{}{\Gamma \cup \{x : \tau'\} \vdash x : \tau'}$$

(ABS) $$\frac{\Gamma \cup \{x : \tau_0\} \vdash E : \tau_1}{\Gamma \vdash \lambda x.E : \tau_0 \to \tau_1}$$

(APP) $$\frac{\Gamma \vdash M : \tau_0 \to \tau_1 \quad \Gamma \vdash N : \tau_0}{\Gamma \vdash (MN) : \tau_1}$$

We call a λ-term E *typed* (or equivalently a term of TLC) and τ' a type of E, if $\Gamma \vdash E : \tau'$ is derivable by the above rules, for some Γ and τ'.

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 $\lambda x.\lambda y.y \rhd_\alpha \lambda x.\lambda z.z$. Variables that are not bound are *free*. We do not distinguish between terms that differ only in the names of bound variables, and we write $e = e'$ to denote the syntactic identity of e and e' *except for the names of their bound variables.*

We write $e \rhd_\beta e'$ (β-reduction) when e' can be derived from e by replacing a subterm in e of the form $(\lambda x.E)E'$, called a *redex*, by $E[x := E']$, E with E' substituted for all free occurrences of x in E; see [5] for standard definitions of substitution and α- and β-reduction.

The *operational semantics* of TLC are defined using *reduction*. Let $\rhd$ be the reflexive, transitive closure of $\rhd_\alpha$ and $\rhd_\beta$. Note that, reduction preserves types.

Curry vs Church Notation: In the above definition, we have adopted the "Curry style" of TLC, where types can be *reconstructed* for unadorned terms using the inference rules. We could have chosen the "Church style," where types and terms are defined together and λ-bound variables are annotated with their type (i.e., we would have $\lambda x : \tau_0.e$ instead of $\lambda x.e$). In our encodings below we will often provide "Church style" type annotations for readability.

TLC$^=$: We obtain TLC$^=$ by enriching TLC 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 rules are the same with one modification: the Γ's must treat the constants as always 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 the reduction rule $\rhd_\delta$ (known as δ-reduction) and consider the additional *redex*:

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

The motivation behind the δ-reduction rule is the desire to express statements of the form "if $x = y$ then p else q"—with the above rule, this can be written simply as $(Eq\, x\, y\, p\, q)$.

The *operational semantics* of TLC$^=$ are defined using *reduction*. Let $\rhd$ be the reflexive, transitive closure of $\rhd_\alpha$, $\rhd_\beta$ and $\rhd_\delta$. Once again, reduction preserves types. A λ-term from which no reduction is possible, i.e., it contains no redex, is in *normal form*.

TLC and TLC$^=$ enjoy the following properties, see [5, 21]:

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

2. *Strong normalization property:* For each e, there exists a nonegative integer i such that if $e \rhd e'$, then the derivation involves no more than i individual reductions.

3. *Principal type property:* A typed λ-term e has a principal type, that is a type from which all other types can be obtained via substitution of types for type variables.

4. *Type reconstruction property:* One can show that given e it is decidable whether e is a typed λ-term and the principal type of this term can be reconstructed. Also, given $\Gamma \vdash e : \tau'$, it is decidable if this statement is derivable by the above rules. (Both these algorithms use first-order unification and reconstruct types. They work with or without type annotations and with or without constants in the Γ's). TLC and TLC$^=$ type reconstruction is *linear-time* in the size of the program analyzed.

For many semantic properties of TLC we refer to [21, 44, 45]. Finally, we refer to [5] for other reduction notions such as the η-reduction, $(\lambda x.\, M\, x) \rhd_\eta M$, where x is not free in M. We sometimes use properties of η-reduction in the proofs of this paper, but do not use $\rhd_\eta$ as part of $\rhd$.

Functionality Order: The *order* of a type, which measures the higher-order functionality of a λ-term of that type, is defined as $\mathrm{order}(t) = 0$ for a type variable t, and $\mathrm{order}(\sigma' \to \sigma'') = \max(1 + \mathrm{order}(\sigma'), \mathrm{order}(\sigma''))$. 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 $\mathtt{o}$ and τ is 0. The above definitions and properties hold for fragments of TLC and TLC$^=$, where the order of terms is bounded by some fixed k. In such fragments we use the above inference rules (Var), (Abs), and (App), but with all types restricted to order k or less.

2.2 Let-Polymorphism: core-ML and core-ML$^=$

The syntax of core-ML$^=$ (core-ML) is that of TLC$^=$ (TLC) augmented with one new term construct: $\mathtt{let}\ x = \mathcal{E}\ \mathtt{in}\ \mathcal{E}$ and a ML-typedness requirement, instead of a typedness requirement.

ML-typedness: This involves the same *monomorphic* types and inference rules for TLC and the constants used for TLC$^=$, with one additional rule that captures some *polymorphism* (see [31]):

$$\text{(Let)} \qquad \frac{\Gamma \vdash E : \tau_0 \quad \Gamma \vdash B\,[x := E] : \tau}{\Gamma \vdash \mathtt{let}\ x = E\ \mathtt{in}\ B : \tau}$$

We call a λ-term E *ML-typed* if $\Gamma \vdash E : \tau'$ is derivable by the inference rules (Var), (Abs), (App), and (Let) for some Γ and τ'. One consequence is that TLC$^=$ (TLC) is a subset of core-ML$^=$ (core-ML).

The operational semantics is as for TLC$^=$ (TLC), where in addition $\mathtt{let}\ x = M\ \mathtt{in}\ N$ is treated as $(\lambda x.\, N)\, M$. A consequence of this is that core-ML$^=$ (core-ML) has the same expressive power as TLC$^=$ (TLC). However, it allows more flexibility in typing.

For example, `let` $x = (\lambda z.\, z)$ `in` $(x\, x)$ is in core-ML but $(\lambda x.\, x\, x)(\lambda z.\, z)$ is not in TLC; the equivalent program in TLC is what we get after one reduction of $(\lambda x.\, x\, x)(\lambda z.\, z)$, namely $(\lambda z.\, z)(\lambda z.\, z)$.

Note that, core-ML$^=$ (core-ML) has all the properties of TLC$^=$ (TLC), i.e., Church-Rosser, Strong Normalization, Principal Type and Type Reconstruction. There is only one difference: Type reconstruction is no longer in linear-time but EXPTIME-complete [31, 32].

2.3 List Iteration

We briefly review how list iteration works. Let $\{e_1, e_2, \ldots, e_k\}$ be a set of λ-terms, each of type α; then

$$L := \lambda c : \alpha \to \beta \to \beta.\, \lambda n : \beta.\, c\, e_1\, (c\, e_2 \ldots (c\, e_k\, n) \ldots)$$

is a λ-term of type $(\alpha \to \beta \to \beta) \to \beta \to \beta$, for *any* type β—in other words, L is a typable term no matter what type β we choose (though one fixed type must be chosen). L is called a *list iterator* encoding the list $e_1, e_2, \ldots, e_k$; the variables c and n abstract over list constructors *Cons* and *Nil*.

To see how such list iterators are used, think of L as a "do"-loop defined by a "loop body" c and a "seed value" n. The loop body is invoked once for every e_i, starting from the last and proceeding backwards to the first. By Church-Rosser and strong normalization, all evaluation orders lead to the same results, so we choose the above one for purposes of presentation. At every invocation, the loop body is provided with two arguments: the current element of the list and an "accumulator" containing the value returned by the previous invocation of the loop body; initially, the accumulator is set to n. From these data, the loop body produces a new value for the accumulator and the iteration continues with the previous element of the list. Once all elements have been processed, the final value of the accumulator is returned.

Booleans: In this paper, we use a standard encoding of Boolean logic where $\mathit{True} := \lambda x : \tau.\, \lambda y : \tau.\, x$ and $\mathit{False} := \lambda x : \tau.\, \lambda y : \tau.\, y$, both of type $\mathsf{Bool} := \tau \to \tau \to \tau$.

Parity: As an example, consider the problem of determining the parity of a list of Boolean values. The exclusive-or function can be written as

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

To compute the parity of a list of Boolean values, we begin with an accumulator value of *False* and then loop over the elements in the list, setting at each stage the accumulator to the exclusive-or of its previous value and the current list element. Thus, the parity function can be written simply as:

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

If L is a list $\lambda c.\, \lambda n.\, c\, e_1\, (c\, e_2 \ldots (c\, e_k\, n) \ldots)$, the term $(\mathit{Parity}\ L)$ reduces to

$$\mathit{Xor}\, e_1\, (\mathit{Xor}\, e_2 \ldots (\mathit{Xor}\, e_k\, \mathit{False}) \ldots),$$

which indeed computes the parity of $e_1, \ldots, e_k$. 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.

Length: As another example, to compute the length of a list, we define

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

where $\mathit{Succ} \equiv \lambda n : (\tau \to \tau) \to \tau \to \tau.\, \lambda s : \tau \to \tau.\, \lambda z : \tau.\, n\, s\, (s\, z)$ and $\mathit{Zero} \equiv \lambda s : \tau \to \tau.\, \lambda z : \tau.\, z$ code successor and zero on the Church numerals (of type $\mathsf{Int} \equiv (\tau \to \tau) \to \tau \to \tau$). The variable x in the "loop body" $\lambda x : \alpha.\, \mathit{Succ}$ serves to absorb the current element of L; the successor function is then applied to the accumulator value.

List iteration is a powerful programming technique, which can be used in the context of TLC and TLC$^=$ to encode any elementary recursion [38, 43]. However, some care is needed if one is to maintain well-typedness (cf. the "type-laundering" technique of [25]).

3 Representing Databases and Queries

3.1 Databases as Lambda Terms

Relations are represented in our setting as simple generalizations of Church numerals. 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 λ-term:*

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

in which every tuple of r appears exactly once. Note that there are many possible encodings, one for each ordering of the tuples in r.

Just like a Church numeral $\lambda s.\, \lambda z.\, s\,(s\,(s \ldots (s\, z)) \ldots)$, the term $\overline{r}$ is a list iterator. The only difference is that $\overline{r}$ 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 an arbitrary type variable. (If r is empty or contains only one tuple, this type is only an instance of the principal type of $\overline{r}$.) 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 θ for σ, are abbreviated as o_k^θ, or, if the exact nature of θ does not matter, as o_k^*. Note that the exponent in this notation denotes the type of the "accumulator value" passed between stages of a list iteration; that is, if $\overline{r}$ is used as an iterator with "accumulator" type θ, then its type must be o_k^θ.

We say that $\overline{r}$ is an encoding *with duplicates* of relation $r \subseteq O^k$ when Definition 3.1 holds with the weaker restriction that every tuple in r appears at least once. This is an interesting variation because the principal type o_k^σ characterizes encodings with duplicates of relations.

Lemma 3.2 *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 $f = \lambda c.\, c\, o_{1,1} \ldots o_{1,k}$ or $f = \overline{r}$ is an encoding with duplicates for some relation $r \subseteq O^k$.*

Proof: Since f does not contain free variables and none of the TLC$^=$ constants has type o_k^q, f must be an abstraction, i.e., $f = \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, f'$, where f' is of type $\tau \to \tau$. There are three possibilities for f': Either $f' = c\, e_1 \ldots e_k$, where $\mathrm{type}(e_i) = \mathsf{o}$ for $1 \leq i \leq k$, or $f' = Eq\, e_1\, e_2\, e_3$, where $\mathrm{type}(e_1) = \mathrm{type}(e_2) = \mathsf{o}$ and $\mathrm{type}(e_3) = \tau$, or $f' = \lambda n : \tau.\, f''$, where $\mathrm{type}(f'') = \tau$.

In the first case, e_i cannot be an abstraction, so it is of the form $x\, t_1 \ldots t_j$, where $j \geq 0$ and $x \in \{c, Eq, o_1, o_2, \ldots\}$. Clearly, $x = c$ and $x = Eq$ are impossible, so each e_i is a constant $o_{1,i}$ and we have $f = \lambda c.\, c\, o_{1,1} \ldots o_{1,k}$.

In the second case, the same argument shows that e_1 and e_2 must be constants, but then $Eq\, e_1\, e_2$ is a redex, so this case is impossible.

In the last case, f'' must have the form $x\, t_1 \ldots t_j$ where $j \geq 0$ and $x \in \{c, n, E\, q, o_1, o_2, \ldots\}$. Clearly, $x = o_i$ is impossible, and $x = Eq$ would produce a redex (t_1 and t_2 would have to be constants), so either $f'' = n$ or $f'' = c\, t_1 \ldots t_k\, t_{k+1}$, where $\mathrm{type}(t_i) = \mathsf{o}$ for $1 \leq i \leq k$ and $\mathrm{type}(t_{k+1}) = \tau$. If $f'' = n$, then $f = \lambda c.\, \lambda n.\, n$ is an encoding of the empty relation. Otherwise, $t_1, \ldots, t_k$ must be constants and the possibilities for t_{k+1} are the same as those for f'', so

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

for some $m \geq 1$. Note that an encoding with duplicate tuples is possible. □

Remark 3.3 *Since the two terms $\lambda c.\, c\, o_{1,1} \ldots o_{1,k}$ and $\lambda c.\, \lambda n.\, c\, o_{1,1} \ldots o_{1,k}\, n$, η-convert (see [5]) 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.*

Note that an encoding of r inherently *orders* the tuples of r. In our setting, we are therefore always dealing with ordered relations, i.e., lists instead of sets of tuples. This has to be taken into account when comparing the expressive power of our languages to more traditional database languages. We define the notion of a *list-represented* database to capture the order implicit in our encodings:

Definition 3.4 *A* list-represented relation *of arity k over universe O is a pair $(r, <)$, where r is a subset of O^k and $<$ is a linear order on r. A* list-represented database *over universe O is a tuple of list-represented relations over O.*

List-represented relations $(r, <)$ are in one-to-one correspondence with relations encoded as $\overline{r}$ according to Definition 3.1 above (i.e., without duplicates). So, when duplicate freedom is clear, we use notation $\overline{r}$ instead of $(r, <)$.

In the following sections, when assessing the expressive power of various query languages, we will always consider computations over list-represented databases, or equivalently, TLC-encoded databases $(\overline{r_1} \ldots \overline{r_l})$. The database *active domain* D is the set of constants in $(\overline{r_1} \ldots \overline{r_l})$.

3.2 Query Languages

We first provide definitions for the FO- and PTIME-queries. These are like the definitions in [2], but over list-represented databases. So the list orders $<_i$ are used instead of some order of the domain of constants of the input finite structure.

Definition 3.5 *A FO-query π of arity $(k_1,\ldots,k_l,k)$ maps list-represented databases $(\overline{r_1}\ldots\overline{r_l})$ of arity $(k_1,\ldots,k_l)$ to relations r of arity k. This mapping is defined by a first-order formula $\psi_\pi(\xi_1,\ldots,\xi_k)$, with free variables $\xi_1,\ldots,\xi_k$, built from predicate symbols $R_1,\ldots,R_l$ of arity $k_1,\ldots,k_l$ and predicate symbols $<_i$ $(1 \le i \le l)$ each $<_i$ specifying a total order among the tuples interpreting R_i. If D is the set of constants in $(\overline{r_1}\ldots\overline{r_l})$ then $\pi(\overline{r_1}\ldots\overline{r_l})$ is relation*
$r = \{(x_1,\ldots,x_k) \in D^k : (D, \overline{r_1}\ldots\overline{r_l})$ *satisfies* $\psi_\pi(x_1,\ldots,x_k)\}$.

Definition 3.6 *A PTIME-query π of arity $(k_1,\ldots,k_l,k)$ maps list-represented databases $(\overline{r_1}\ldots\overline{r_l})$ of arity $(k_1,\ldots,k_l)$ to relations r of arity k. This mapping is defined by Turing Machine M_π, which runs in time polynomial in the size of its input. If the input $x_1,\ldots,x_k,\overline{r_1}\ldots\overline{r_l}$ is presented as a binary string and if D is the set of constants in $(\overline{r_1}\ldots\overline{r_l})$ then $\pi(\overline{r_1}\ldots\overline{r_l})$ is relation*
$r = \{(x_1,\ldots,x_k) \in D^k : M_\pi(x_1,\ldots,x_k,\overline{r_1}\ldots\overline{r_l})$ *accepts* $\}$.

We now define our query languages. For purposes of comparison we use the same syntax in both TLI and MLI definitions. Note that, in MLI we interpret the outermost λ's as `let`'s.

Definition 3.7 *A query term of arity $(k_1,\ldots,k_l,k)$ in* $\mathrm{TLI}_i^=$ *(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 $\mathtt{o}_k^\sigma$ (where σ is some type variable different from* `o`*).*

Definition 3.8 *A query term of arity $(k_1,\ldots,k_l,k)$ in* $\mathrm{MLI}_i^=$ *(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 type $(\lambda R_1\ldots\lambda R_l.M)\overline{r_1}\ldots\overline{r_l}$ as $\mathtt{o}_k^\sigma$ (where σ is some type variable different from* `o`*) with the bindings $\lambda R_1\ldots\lambda R_l$ typed as* `let`*'s.*

The motivation behind these definitions is the desire to enforce proper input-output behavior of query terms. That is, whenever Q is a query term of arity $(k_1,\ldots,k_l,k)$ and $\overline{r_1},\ldots,\overline{r_l}$ are encodings of relations of arities $k_1,\ldots,k_l$, then the term $(Q\,\overline{r_1}\ldots\overline{r_l})$ should reduce to a normal form that is an encoding (with duplicates) of a relation of arity k. By virtue of Q being a query term, the term $(Q\,\overline{r_1}\ldots\overline{r_l})$ can be typed as $\mathtt{o}_k^\sigma$, where σ is some type variable different from `o`, and then Lemma 3.2 implies that its normal form must be the encoding with duplicates of a relation of arity k.

The above definitions are phrased semantically because they involve quantification over all inputs. However, it is easy to see that they really describe a syntactic property:

Lemma 3.9 *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 decide syntactically if it is a query of $\mathrm{TLI}_i^=$ or $\mathrm{MLI}_i^=$.*

Proof: Any encoding $\overline{r}$ of a k-ary relation r can be typed as $\mathtt{o}_k^\sigma = (\mathtt{o}\to\cdots\to\mathtt{o}\to\sigma\to\sigma)\to\sigma\to\sigma$ (where σ is any type variable), and this type is in fact the principal type if r contains at least two tuples. Thus, $(\lambda R_1\ldots\lambda R_l.M)$ is a $\mathrm{TLI}_i^=$ query term if and only if the term $(\lambda R_1\ldots\lambda R_l.M)x_1\ldots x_l$ types, where $x_1,\ldots,x_l$ are some terms of principal type $\mathtt{o}_{k_1}^{\sigma_1},\ldots,o_{k_l}^{\sigma_l}$, and it is an $\mathrm{MLI}_i^=$ query term if and only if the term $M[R_1 := x_1,\ldots,R_l := x_l]$ types. Decidability follows from the type reconstruction property of TLC$^=$ and core-ML$^=$. □

We, thus, have the classes of database queries:

Definition 3.10 *A $\mathrm{TLI}_i^=(\mathrm{MLI}_i^=)$-query π of arity $(k_1,\ldots,k_l,k)$ maps list-represented databases $(\overline{r_1}\ldots\overline{r_l})$ of arity $(k_1,\ldots,k_l)$ to relations r of arity k. This mapping is defined by a $\mathrm{TLI}_i^=(\mathrm{MLI}_i^=)$-query term Q_π, where $(Q_\pi\,\overline{r_1}\ldots\overline{r_l})$ reduces to an encoding (with duplicates) of r.*

Types of Inputs and Outputs: Note that, unlike [18, 42], we allow the input and output type of a query to differ. Outputs are always typed as o_k^σ, but inputs can be typed as o_k^*, where the $*$ denotes some type that, for query terms in $\mathrm{TLI}_i^=/\mathrm{MLI}_i^=$, can be of order up to i. This convention is necessary for expressing all of PTIME. In fact, the expressive power of the $\mathrm{TLI}_i^=/\mathrm{MLI}_i^=$ languages comes directly from the ability to use the input relations as iterators over higher-order "accumulator" values. From the definition, it is easy to see that a $\mathrm{TLI}_i^=/\mathrm{MLI}_i^=$ query term can use its inputs as iterators over "accumulator" values of order up to i.

To simplify the subsequent discussion, we assume from now on that all typings use only the distinct type variables o and τ, where o denotes the type of constants $o_1, o_2, \ldots$ and τ is the fixed variable chosen for the typing $Eq : \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$. In particular, we type encodings of relations as o_k^τ or o_k^θ, where θ is a type over o and τ. Clearly, this does not lead to any loss of generality, because any term typable at all has a type involving only o and τ, which can be obtained from its principal type by substituting τ for all type variables different from o.

4 Lower Bounds on Expressibility

To illustrate the power of list iteration, let us show how to express various well-known database queries in $\mathrm{TLI}_0^=$ and $\mathrm{TLI}_1^=$. We begin by coding relational operators in $\mathrm{TLI}_0^=$. The techniques are presented in full detail (albeit under slightly different input-output conventions) in [25] and we just give the $\mathit{Intersection}_k$ operator as an example here, for relations of arity k. For reference, the coding of the other relational operators is given in the Appendix.

We first need terms Equal_k to test for equality of k-tuples and Member_k to test for membership of a k-tuple in a k-ary relation. $(\mathit{Equal}_k\, x_1 \ldots x_k\, y_1 \ldots y_k)$ reduces to $\mathit{True} = \lambda u.\, \lambda v.\, u$ if the tuples $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_k)$ are equal and to $\mathit{False} = \lambda u.\, \lambda v.\, v$ otherwise. Similarly, $(\mathit{Member}_k\, x_1 \ldots x_k\, \overline{r})$ reduces to True if the tuple $(x_1, \ldots, x_k)$ is a member of the relation r and to False otherwise.

$$
\begin{aligned}
&\mathit{Equal}_k : \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \mathsf{Bool} := \\
&\quad \lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o}.\, \lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o}. \\
&\qquad \lambda u : \tau.\, \lambda v : \tau. \\
&\qquad\quad Eq\, x_1\, y_1\, (Eq\, x_2\, y_2 \ldots (Eq\, x_k\, y_k\, u\, v)\, v) \ldots v
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Member}_k : \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \mathsf{o}_k^\tau \to \mathsf{Bool} := \\
&\quad \lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o}.\, \lambda R : \mathsf{o}_k^\tau. \\
&\qquad \lambda u : \tau.\, \lambda v : \tau. \\
&\qquad\quad R\, (\lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o}.\, \lambda T : \tau.\, (\mathit{Equal}_k\, x_1 \ldots x_k\, y_1 \ldots y_k)\, u\, T)\, v
\end{aligned}
$$

With the aid of these terms, $\mathit{Intersection}_k$ can be coded as follows:

$$
\begin{aligned}
&\mathit{Intersection}_k : \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau := \\
&\quad \lambda R : \mathsf{o}_k^\tau.\, \lambda S : \mathsf{o}_k^\tau. \\
&\qquad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, \lambda n : \tau. \\
&\qquad\quad R\, (\lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o}.\, \lambda T : \tau.\, (\mathit{Member}_k\, x_1 \ldots x_k\, S)\, (c\, x_1 \ldots x_k\, T)\, T)\, n
\end{aligned}
$$

By inspection of its type, $\mathit{Intersection}_k$ is a $\mathrm{TLI}_0^=$ query term and it is easy to verify that it indeed computes the intersection of its input relations.

Another example is a term $Order_k$ with the property that $(Order_k\, x_1 \ldots x_k\, y_1 \ldots y_k\, \overline{r})$ reduces to *True* if tuple $(x_1, \ldots, x_k)$ precedes tuple $(y_1, \ldots, y_k)$ in $\overline{r}$ and to *False* otherwise. $Order_k$ can be coded as follows:

$$
\begin{array}{l}
Order_k : \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \mathsf{o}_k^\tau \to \mathsf{Bool} := \\
\quad \lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o}.\, \lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o}.\, \lambda R : \mathsf{o}_k^\tau. \\
\qquad \lambda u : \tau.\, \lambda v : \tau. \\
\qquad\quad R\,(\lambda z_1 : \mathsf{o} \ldots \lambda z_k : \mathsf{o}.\, \lambda T : \tau.\, Equal_k\, x_1 \ldots x_k\, z_1 \ldots z_k\, u\,(Equal_k\, y_1 \ldots y_k\, z_1 \ldots z_k\, v\, T))\, v
\end{array}
$$

These encodings, together with Codd's equivalence theorem for relational algebra and calculus, establish the following theorem (first shown in [25]):

Theorem 4.1 *Every FO-query, over list-represented databases, is a $TLI_0^=$ ($MLI_0^=$)-query.*

As a next step, we illustrate how to encode fixpoint queries in $\mathrm{TLI}_1^=$. The technique is presented in detail in [25] and we just review the main steps here.

Let $\lambda R_1 \ldots \lambda R_l\, \lambda R.\, M$ be a $\mathrm{TLI}_0^=$ query term such that for given relations $r_1, \ldots, r_l$, the term $Q := \lambda R.\, M\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$ denotes a monotone mapping from k-ary relations to k-ary relations. To compute the minimal fixpoint of Q's mapping, it suffices to iterate Q a polynomial number of times starting from the empty relation. This can be done by constructing a sufficiently long list from the input relations and then using that list as an iterator to apply Q polynomially many times to an encoding of the empty relation. Thus, in principle, the encoding of a fixpoint query looks like:

$$Fix = \lambda R_1 \ldots \lambda R_l.\, Crank\,(\lambda R.\, M)\,(\lambda c.\, \lambda n.\, n)$$

where *Crank* is a sufficiently large Church numeral of the form $Length\,(D \times \cdots \times D)$, and D, the *active domain* of the input database, is computed by a sequence of projections and unions using the $\mathrm{TLI}_0^=$ implementations of the relational operators (see Appendix).

There are two technical difficulties that need to be overcome under this approach. One involves intermediate representation and the other typing.

Intermediate Representation: First, the intermediate results that are passed from one stage of the fixpoint computation to the next are relations, which, in their list representation, are order 2 objects. In $\mathrm{TLI}_1^=$, however, iterations can pass only order 1 objects between stages. Therefore, another representation of relations has to be devised that requires only an order 1 type. The solution here is to represent relations by their *characteristic* functions. The characteristic function f_r of a k-ary relation r is a $\mathrm{TLC}^=$ term of type $\chi_k := \mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{Bool}$, such that for any k constants $o_{i_1}, \ldots, o_{i_k}$,

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

Using the active domain D of the input database computed earlier, it is possible to write λ-terms *FuncToList* and *ListToFunc* that translate between the list and characteristic function representation of a relation.

$$
\begin{array}{l}
ListToFunc : \mathsf{o}_k^\tau \to \chi_k := \\
\quad \lambda R : \mathsf{o}_k^\tau. \\
\qquad \lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o}. \\
\qquad\quad \lambda u : \tau.\, \lambda v : \tau. \\
\qquad\qquad R\,(\lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o}.\, \lambda T : \tau.\, Equal_k\, x_1 \ldots x_k\, y_1 \ldots y_k\, u\, T)\, v.
\end{array}
$$

$$
\begin{array}{l}
\mathit{FuncToList} : \chi_k \to \mathsf{o}_k^\tau := \\
\quad \lambda f : \mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{Bool}. \\
\quad\quad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau. \lambda n : \tau. \\
\quad\quad\quad D\,(\lambda x_1 : \mathsf{o}. \lambda T_1 : \tau. \\
\quad\quad\quad\quad D\,(\lambda x_2 : \mathsf{o}. \lambda T_2 : \tau. \\
\quad\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad\quad\quad D\,(\lambda x_k : \mathsf{o}. \lambda T_k : \tau. \\
\quad\quad\quad\quad\quad\quad\quad f\, x_1 \ldots x_k\,(c\, x_1 \ldots x_k\, T_k)\, T_k)\, T_{k-1})\ldots T_1)\, n,
\end{array}
$$

With these operators, the above query can be rewritten as

$$
\mathit{Fix} := \lambda R_1 \ldots \lambda R_l.\, \mathit{FuncToList}\,(\mathit{Crank}\,(\lambda f.\, \mathit{ListToFunc}\,((\lambda R.\, M)\,(\mathit{FuncToList}\, f)))\,(\lambda \vec{x}.\, \mathit{False})),
$$

which passes only order 1 objects in the "accumulator".

Typing Constraints: Another difficulty arises in connection with the typing of the input relations. Inside M, the inputs $R_1, \ldots, R_l$ are used in $\mathrm{TLI}_0^=$-encoded relational operators, and in this context, they need to be typed as $\mathsf{o}_{k_1}^\tau, \ldots, \mathsf{o}_{k_l}^\tau$ (i.e., iterators with a base "accumulator" type). However, *Crank* is used to iterate over characteristic functions, which have an order 1 type $\chi := \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau \to \tau$, so the occurrences of $R_1, \ldots, R_l$ in *Crank* need to be typed as $\mathsf{o}_{k_1}^\chi, \ldots, \mathsf{o}_{k_l}^\chi$ (i.e., iterators with an order 1 "accumulator" type). These typings do not unify, so in order to type the *Fix* query as written above, it is necessary to use `let`-polymorphism. By interpreting the λ-bindings of $R_1, \ldots, R_l$ as `let`-bindings, each occurrence of R_i in *Fix* can be typed independently, and the term becomes typable. Thus, *Fix* in its form above is a $\mathrm{MLI}_1^=$ query.

However, it is possible to modify *Fix* further to obtain a $\mathrm{TLI}_1^=$ query term. The crucial step is the introduction of "type-laundering" gadgets. These are a family Copy_i, $1 \le i \le l$, of $\mathrm{TLC}^=$ terms such that $(\mathit{Copy}_i\, R_i)$ reduces to a copy of R_i (i.e., another encoding of the same relation), but in such a way that R_i can be typed as $\mathsf{o}_{k_i}^\chi$, whereas $(\mathit{Copy}_i\, R_i)$ has type $\mathsf{o}_{k_i}^\tau$. Using these gadgets, the occurrences of R_i in *Crank* can be typed as $\mathsf{o}_{k_i}^\chi$, as required, while all occurrences of R_i in M (and *ListToFunc* and *FuncToList*) can be replaced by the term $(\mathit{Copy}_i\, R_i)$, which has the correct type $\mathsf{o}_{k_i}^\tau$. The construction of the Copy_i operator is described in detail in [25] and its code is in the Appendix.

The final $\mathrm{TLI}_1^=$ version of the fixpoint query becomes

$$
\begin{array}{l}
\mathit{Fix} : \mathsf{o}_{k_1}^\chi \to \cdots \to \mathsf{o}_{k_l}^\chi \to \mathsf{o}_k^\tau := \\
\quad \lambda R_1 : \mathsf{o}_{k_1}^\chi \ldots \lambda R_l : \mathsf{o}_{k_l}^\chi. \\
\quad\quad \mathit{FuncToList}'\,(\mathit{Crank}\,(\lambda f : \chi.\, \mathit{ListToFunc}'\,((\lambda R.\, M')\,(\mathit{FuncToList}'\, f)))\,(\lambda \vec{x} : \vec{\mathsf{o}}.\, \mathit{False})),
\end{array}
$$

where $\mathit{FuncToList}'$ stands for $\mathit{FuncToList}\,[R_1 := (\mathit{Copy}_1\, R_1), \ldots, R_l := (\mathit{Copy}_l\, R_l)]$ and $\mathit{ListToFunc}'$ and M' are defined analogously.

Over ordered databases (in particular list-represented databases), fixpoint queries are sufficient to express all PTIME queries [28, 46], so we have the following theorem (first shown in [25]):

Theorem 4.2 *Every PTIME-query, over list-represented databases, is a $\mathrm{TLI}_1^=$ ($\mathrm{MLI}_1^=$)-query.*

5 Upper Bounds on Expressibility

In this section, we show that the converses of Theorems 4.1 and 4.2 also hold:

Theorem 5.1 *Every $TLI_0^=$ ($MLI_0^=$)-query is a FO-query, over list-represented databases.*

Theorem 5.2 *Every $TLI_1^=$ ($MLI_1^=$)-query is a PTIME-query, over list-represented databases.*

Thus, $TLI_0^=$ and $MLI_0^=$ capture exactly the first-order queries over list-represented databases, and $TLI_1^=$ and $MLI_1^=$ capture exactly the PTIME queries.

To prove these results, we first analyze the structure of $TLI_i^=$ and $MLI_i^=$ query terms based on their input-output behavior and then develop algorithms for evaluating such terms efficiently. For $i = 0$ the evaluation is by translation into a first-order formula. For $i = 1$ we develop a semantic evaluation.

In the following, let Q be a fixed $TLI_i^=$ or $MLI_i^=$ query 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 evaluation, i.e., we assume that the query term is of fixed size and the input data is of variable size and the preprocessing happens before evalution. This eliminates all redexes in Q, as well as, all `let`'s in Q except the outermost ones.

For $MLI_i^=$, we can eliminate all `let`'s from Q by replacing every subterm of the form "`let` $x = N$ `in` M" with $M\,[x := N]$ and by agreeing that variables corresponding to input relations are to be polymorphically typed. This allows us to use essentially identical proofs for $MLI_i^=$ and $TLI_i^=$.

5.1 The Structure of $TLI_i^=$ and $MLI_i^=$ Terms

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 $(T\ S)$. 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.

Since Q is in $TLI_i^=$ or $MLI_i^=$, it can be typed as $\mathtt{o}_{k_1}^* \to \cdots \to \mathtt{o}_{k_l}^* \to \mathtt{o}_k^\tau$, where the asterisks stand for unspecified types of order $\leq i$ built from $\mathtt{o}$ and τ. (In the case of $MLI_i^=$ terms, these types are to be interpreted polymorphically, i.e., we replace each occurrence R_i with the corresponding input and might type each occurrence differently). We call any such typing a *canonical typing* of Q. Under a canonical typing, every subterm t of Q is assigned a certain type, which we call the *canonical type* of t for this canonical typing of Q.

In order to simplify the evaluation of a query, 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 algorithms operate on. A normal form is *closed* if it contains no free variables.

Definition 5.3 *Let Q be a $TLI_i^=$ or $MLI_i^=$ query term and γ be a canonical typing of Q. We say that Q is in* γ-canonical form *if Q is in closed normal form and every complete subterm t of Q is of the form $\lambda x_1 : \alpha_1 \ldots \lambda x_k : \alpha_k.\, M$, where $k \geq 0$ and $\alpha_1, \ldots, \alpha_k$ are such that the canonical type of t under γ is $\alpha_1 \to \cdots \to \alpha_k \to \sigma$, where σ is either τ or $\mathtt{o}$. (This is also known as a long normal form.) We say that Q is in canonical form if it is in γ-canonical form for some canonical typing γ.*

Lemma 5.4 *Let P be a $TLI_i^=$ or $MLI_i^=$ term mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then there is a $TLI_i^=$ or $MLI_i^=$ term Q in canonical form such that P and Q define the same database query, i. e., for every 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, with duplicates, the same relation r. Q can be effectively determined from P.*

Proof: Fix some canonical typing γ of P. We obtain Q from P by a series of η-expansions [5], where a complete subterm $\lambda x_1 : \alpha_1 \ldots \lambda x_k : \alpha_k . M$ with $\mathrm{type}(M) = \alpha_{k+1} \to \cdots \to \alpha_m \to \mathsf{o}$ or $\mathrm{type}(M) = \alpha_{k+1} \to \cdots \to \alpha_m \to \tau$ is replaced by $\lambda x_1 : \alpha_1 \ldots \lambda x_m : \alpha_m . (M\, x_{k+1} \ldots x_m)$, until no further expansions are possible. To see that this does not change the semantics of the query defined by P, fix inputs $\overline{r_1}, \ldots, \overline{r_l}$ and consider the normal form of $(P\,\overline{r_1}\ldots\overline{r_l})$, which is an encoding $\overline{r}$ of a relation r. We have a reduction path

$$(Q\,\overline{r_1}\ldots\overline{r_l}) \xrightarrow{\eta} \cdots \xrightarrow{\eta} (P\,\overline{r_1}\ldots\overline{r_l}) \xrightarrow{\beta,Eq} \cdots \xrightarrow{\beta,Eq} \overline{r}.$$

It is known that in a $\beta\eta$-reduction sequence, η-reductions can always be pushed to the end [5, Theorem 15.1.6]. This remains true even if Eq-reductions are added, because Eq- and η-reductions commute for typed terms, as a case analysis shows. Thus, for some term t of type o_k^τ, we have

$$(Q\,\overline{r_1}\ldots\overline{r_l}) \xrightarrow{\beta,Eq} \cdots \xrightarrow{\beta,Eq} t \xrightarrow{\eta} \cdots \xrightarrow{\eta} \overline{r}.$$

However, it is easy to see that any term t of type o_k^τ that η-reduces to $\overline{r}$ also β-reduces to $\overline{r}$, so it follows that $\overline{r}$ is the β, Eq-normal form of $(Q\,\overline{r_1}\ldots\overline{r_l})$ and hence Q and P define the same query.

If Q is not closed, its free variables can be eliminated by the following procedure: Any occurrence of a free variable F of type, say, $\alpha_1 \to \cdots \to \alpha_k \to \mathsf{o}$ or $\alpha_1 \to \cdots \to \alpha_k \to \tau$, is replaced by a term $\lambda x_1 : \alpha_1 \ldots \lambda x_k : \alpha_k . o_1$ or $\lambda x_1 : \alpha_1 \ldots \lambda x_k : \alpha_k . n$, respectively (where n is the variable of type τ bound at the top level of Q), until no more free variables remain. Since the output of a query does not contain free variables, this procedure does not affect the semantics of Q. After a final normalization, we obtain the desired canonical form. □

Lemma 5.5 *Let Q be a $TLI_i^=$ or $MLI_i^=$ term in canonical form mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then the following are true:*

1. *Q has form $\lambda R_1 : \mathsf{o}_{k_1}^* \ldots \lambda R_l : \mathsf{o}_{k_l}^* . \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau . \lambda n : \tau . Q'$ with $\mathrm{type}(Q') = \tau$.*

2. *Every occurrence of R_i in Q' is of the form $R_i\,(\lambda \vec{x} . \lambda f . \lambda \vec{y} . M)\,(\lambda \vec{z} . N)\,\vec{T}$, where $\vec{x}$ is a vector of k_i variables of type o, f is a variable of order $\leq i$, $\vec{y}$ is a (possibly empty) vector of variables of order $< i$, M and N are terms of type o or τ, and $\vec{T}$ is a (possibly empty) vector of terms of order $< i$. We call f the* accumulator variable *for this occurrence of R_i.*

3. *Every occurrence of Eq in Q' is of the form $Eq\,S\,T\,U\,V$, where S and T are terms of type o and U and V are terms of type τ.*

4. *Every occurrence of c in Q' is of the form $c\,T_1 \ldots T_k\,T_{k+1}$, where the type of $T_1, \ldots, T_k$ is o and the type of T_{k+1} is τ.*

5. *Every occurrence of n in Q' is argument-free, i.e., there are no occurrences of the form $(n\,t)$, with t some term.*

6. *The only free variables in Q' are $R_1, \ldots, R_l$, c, and n.*

7. *The only bound variables in Q' of order i or more are accumulator variables.*

Proof: Items 1–5 follow immediately from the type of Q and the fact that Q is canonical, i. e., all possible η-expansions have been performed. Item 6 follows because canonical forms are closed. To prove Item 7, pick a bound variable y of Q' of order $\geq i$ and assume first that its order is maximal among all bound variables of Q'. Let t be the smallest complete subterm of Q' containing the binding occurrence of y. Then t is of the form $\lambda\vec{x}.\,\lambda y.\,\lambda\vec{z}.\,M$. Since t abstracts on y, its order must be at least $\mathrm{order}(y)+1 > i$. It follows that t cannot be identical to Q', and hence it must be a proper subterm of Q' with some variable x governing its occurrence. Since t is an argument of x, the order of x must be at least $\mathrm{order}(y)+2$, and since y was chosen to be of maximal order among the bound variables of Q', x must be free in Q'. Item 6 then implies that x is one of $R_1, \ldots, R_l$, and because $\mathrm{order}(y) \geq i$, Item 2 implies that y is an accumulator variable. Moreover, since accumulator variables in $\mathrm{TLI}_i^=$ query terms can have order at most i, it follows that $\mathrm{order}(y) = i$, so i is the maximal order of bound variables of Q' and each bound variable of that order is an accumulator variable. □

5.2 Evaluating $\mathrm{TLI}_0^=$ and $\mathrm{MLI}_0^=$ Terms

Overview of Evaluation: $\mathrm{TLI}_0^=/\mathrm{MLI}_0^=$ query terms can only perform iterations where the type of the "accumulator" is of order 0, i.e., τ or $\mathtt{o}$. Interestingly, iterations of this kind are not truly sequential—instead, all their stages can be evaluated *in parallel*, producing the output of the query in constant parallel time, or equivalently [29], in first-order logic.

The intuition behind this is the following. It is impossible for a $\mathrm{TLC}^=$ term to distinguish between any two arguments of type τ. Thus, in an iteration with an "accumulator" of type τ, the processing at each stage cannot depend on the incoming value—either the incoming value is ignored altogether, or it is passed on unchanged (perhaps as part of a larger structure) to the next stage. It is possible to construct first-order formulas that describe whether a stage ignores its input and if not, what kind of data it adds to it before passing it on to the next stage. The output of an iteration can then be described by a formula that says: "something is in the output if (a) it was in the initial value of the accumulator and none of the iteration stages ignored its input, or (b), it was produced at some stage and none of the later stages ignored its input." Note that the concept of "later stages" is first-order expressible over list-presented databases, since they come with an ordering of the tuples in each relation.

The situation is similar, albeit slightly more complicated, for iterations with an "accumulator" of type $\mathtt{o}$. Here, a stage could conceivably "look" at the incoming value by means of the Eq predicate. However, Eq has type $\mathtt{o} \to \mathtt{o} \to \tau \to \tau \to \tau$, so any Eq term eventually reduces to a term of type τ, whereas the stage needs to produce an "accumulator" value of type $\mathtt{o}$. Thus, Eq cannot be used to compute the new "accumulator" value (here we use the fact that $\mathtt{o}$ and τ are different variables), and hence, just as in the case above, the stage must be oblivious to the incoming value. The same techniques can then be applied to construct a first-order formula describing the output of the iteration.

Query Term Structure: Before we describe the construction in detail, let us investigate the structure of $\mathrm{TLI}_0^=/\mathrm{MLI}_0^=$ query terms some more. This is useful because we will do induction on subterms of type τ and $\mathtt{o}$.

Lemma 5.6 *Let $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ be a $TLI_0^=$ or $MLI_0^=$ term in canonical form mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then every subterm of Q of type τ has one of the following forms:*

1. *$R_i\,(\lambda x_1 : \mathtt{o} \ldots \lambda x_{k_i} : \mathtt{o}.\,\lambda y : \tau.\,M)\,N$, where M and N are terms of type τ,*

2. *$Eq\,S\,T\,U\,V$, where S and T are terms of type o and U and V are terms of type τ,*

3. *$c\,T_1 \ldots T_k\,T_{k+1}$, where $T_1, \ldots, T_k$ are terms of type o and T_{k+1} is a term of type τ,*

4. *y, where y is either an accumulator variable of type τ or $y = n$.*

Every subterm of Q of type o has one of the following forms:

5. *$R_i\,(\lambda x_1 : \mathsf{o} \ldots \lambda x_{k_i} : \mathsf{o}.\, \lambda y : \mathsf{o}.\, M)\,N$, where M and N are terms of type o,*

6. *y, where y is an accumulator variable of type o,*

7. *o_j, where o_j is a $TLC^{=}$ constant.*

Proof: Let M be a subterm of Q of type τ and let s be its top-level symbol, i. e., $M = s\,M_1 \ldots M_n$. s must be one of R_i, Eq, c, n, or an accumulator variable of type τ, because Lemma 5.5 implies that no other variables can occur in Q. For each of these cases, it follows from Lemma 5.5 that one of (1)–(4) must apply.

If M is of type o, then its top-level symbol cannot be Eq, c, or n. Thus, it must either be an R_i, in which case (5) applies, or a variable of type o or an explicit constant, in which case (6) or (7) apply. □

First-Order Evaluation: For a term Q as described in the above lemma, we will now construct a first-order formula $\psi_Q\,(\xi_1, \ldots, \xi_k)$ describing its behavior. More precisely, $\psi_Q\,(\xi_1, \ldots, \xi_k)$ is a formula with free variables $\xi_1, \ldots, \xi_k$ built from predicate symbols $R_1, \ldots, R_l$ of arities $k_1, \ldots, k_l$ and interpreted predicates $Precedes_i$ $(1 \leq i \leq l)$ specifying a total order among the tuples in R_i, such that for any structure $\mathcal{S} = (D, \overline{r_1}, \ldots, \overline{r_l})$ over $R_1, \ldots, R_l$ and any tuple $(x_1, \ldots, x_k) \in D^k$, we have $\mathcal{S} \vdash \psi_Q\,(x_1, \ldots, x_k)$ iff $(x_1, \ldots, x_k)$ is in the output of $(Q\,\overline{r_1} \ldots \overline{r_l})$, where encoding $\overline{r_i}$ is chosen so that the tuples appear in the order determined by $Precedes_i$. (By $\vdash$ we mean satisfies).

Let us first describe the construction of ψ_Q informally. The main task is to describe the output of an iteration $R_i\,(\lambda \vec{x}.\, \lambda y.\, M)\,N$, where y, M, and N are of type τ. It is easy to see that during the evaluation of $(Q\,\overline{r_1} \ldots \overline{r_l})$, such an iteration must eventually reduce to a term

$$
\begin{aligned}
L = c\,o_{1,1}\,o_{1,2} \ldots o_{1,k} \\
\quad (c\,o_{2,1}\,o_{2,2} \ldots o_{2,k} \\
\quad \ldots \\
\quad\quad (c\,o_{m,1}\,o_{m,2} \ldots o_{m,k}\,z)) \ldots),
\end{aligned}
$$

where $m \geq 0$ and z is some variable of type τ. Since each stage of the iteration cannot "look" at the value that is passed in, it can only either prepend some tuples to the incoming value or throw it away altogether and start building a new list from scratch. Thus, a tuple can end up in L in two ways: either it was already present in the initial value N of the iteration and every stage of the iteration only added tuples to the incoming value, or it was contributed at some stage and all subsequent stages only added tuples to the incoming value.

To capture the output of an iteration in a first-order formula, we therefore need to express two things: (a) a stage of the iteration prepends tuples to the incoming value, i. e., it "passes through" all incoming tuples to the next stage, and (b) a stage of the iteration "produces" some tuple $(\xi_1, \ldots, \xi_k)$. This leads to the definition of two sets of formulas: a formula $PassThrough_{z,t}$ for every subterm $t : \tau$ of Q and variable $z : \tau$ free in t, saying that term t will "pass through" whatever tuples are in z to its output, and a formula $Produces_t\,(\xi_1, \ldots, \xi_k)$ for every subterm $t : \tau$

of Q saying that tuple $(\xi_1, \ldots, \xi_k)$ will be in the output of t. These formulas are defined below by structural induction over t. The formula $Produces_{Q'}(\xi_1, \ldots, \xi_k)$, where Q' is the "body" of $Q = \lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$, is then the desired first-order equivalent of Q.

The approach described above has to be slightly modified to deal with iterations of the form $R_i\,(\lambda \vec{x}.\, \lambda y.\, M)\,N$, where y, M, and N are of type o. Such iterations reduce eventually to a single constant. The equivalent of the *PassThrough* and *Produces* formulas for this case are a formula $PassThrough'_{z,t}$ for every subterm $t : \mathsf{o}$ of Q and variable $z : \mathsf{o}$ free in t, saying that term t will "pass through" whatever constant is in z to its output, and a formula $Produces'_t(\xi)$ for every subterm $t : \mathsf{o}$ of Q saying that ξ is the output of t. These formulas are also defined by structural induction over t. Interestingly, the formula $PassThrough'_{z,t}$ is a *closed* formula, i. e., it does not depend on the values of any free variables of t. In other words, the behavior of a "loop body" of type $\mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{o}$ is *oblivious* to its arguments. This is because Eq cannot appear in such loop bodies; the only conditional terms that can appear are of the form $\lambda u.\, \lambda v.\, R_i\,(\lambda \vec{x}.\, \lambda y.\, u)\,v$, which test the emptyness of relation R_i.

The First-Order Formulas: Here are the actual definitions of the *PassThrough* and *Produces* predicates. We show in Lemma 5.7 below that they have indeed the desired properties.

$$
\begin{aligned}
PassThrough_{z,x} &:= \begin{cases} False & \text{if } z \neq x \\ True & \text{if } z = x \end{cases} \quad (\text{where } x : \tau \text{ is a variable}) \\
PassThrough_{z,cT_1 \ldots T_k T_{k+1}} &:= PassThrough_{z,T_{k+1}} \\
PassThrough_{z,EqSTUV} &:= \exists x y\ Produces'_S(x) \wedge Produces'_T(y) \\
&\quad \wedge \Big((x = y \wedge PassThrough_{z,U}) \\
&\qquad \vee (x \neq y \wedge PassThrough_{z,V}) \Big) \\
PassThrough_{z,R_i(\lambda \vec{x}.\lambda y.M)N} &:= \Big(PassThrough_{z,N} \wedge \forall \vec{x} \in R_i\ \ PassThrough_{y,M} \Big) \\
&\quad \vee \Big(\exists \vec{x}_0 \in R_i\ \ PassThrough_{z,M[\vec{x}:=\vec{x}_0]} \\
&\qquad \wedge \forall \vec{x} \in R_i\ \ Precedes_i(\vec{x}, \vec{x}_0) \Rightarrow PassThrough_{y,M} \Big)
\end{aligned}
$$

$$
\begin{aligned}
Produces_x(\xi_1, \ldots, \xi_k) &:= False \quad (\text{where } x : \tau \text{ is a variable}) \\
Produces_{cT_1 \ldots T_k T_{k+1}}(\xi_1, \ldots, \xi_k) &:= \left(\bigwedge_{i=1}^{k} Produces'_{T_i}(\xi_i) \right) \vee Produces_{T_{k+1}}(\xi_1, \ldots, \xi_k) \\
Produces_{EqSTUV}(\xi_1, \ldots, \xi_k) &:= \exists x y\ Produces'_S(x) \wedge Produces'_T(y) \\
&\quad \wedge ((x = y \wedge Produces_U(\xi_1, \ldots, \xi_k)) \\
&\qquad \vee (x \neq y \wedge Produces_V(\xi_1, \ldots, \xi_k))) \\
Produces_{R_i(\lambda \vec{x}.\lambda y.M)N}(\xi_1, \ldots, \xi_k) &:= \Big(Produces_N(\xi_1, \ldots, \xi_k) \wedge \forall \vec{x} \in R_i\ \ PassThrough_{y,M} \Big) \\
&\quad \vee \Big(\exists \vec{x}_0 \in R_i\ \ Produces_{M[\vec{x}:=\vec{x}_0]}(\xi_1, \ldots, \xi_k) \\
&\qquad \wedge \forall \vec{x} \in R_i\ \ Precedes_i(\vec{x}, \vec{x}_0) \Rightarrow PassThrough_{y,M} \Big)
\end{aligned}
$$

$$
\begin{aligned}
PassThrough'_{z,o_j} &:= \mathit{False} \quad (\text{where } o_j \text{ a constant}) \\
PassThrough'_{z,x} &:= \begin{cases} \mathit{False} & \text{if } z \neq x \\ \mathit{True} & \text{if } z = x \end{cases} \quad (\text{where } x : \mathsf{o} \text{ is a variable}) \\
PassThrough'_{z,R_i(\lambda\vec{x}.\lambda y.M)N} &:= \Big(PassThrough'_{z,N} \wedge \neg\exists\vec{x} \in R_i\Big) \\
&\quad \vee \Big(PassThrough'_{z,N} \wedge PassThrough'_{y,M} \wedge \exists\vec{x} \in R_i\Big) \\
&\quad \vee \Big(PassThrough'_{z,M} \wedge \exists\vec{x} \in R_i\Big)
\end{aligned}
$$

$$
\begin{aligned}
Produces'_{o_j}(\xi) &:= (\xi = o_j) \quad (\text{where } o_j \text{ is a constant}) \\
Produces'_{x}(\xi) &:= (\xi = x) \quad (\text{where } x : \mathsf{o} \text{ is a variable}) \\
Produces'_{R_i(\lambda\vec{x}.\lambda y.M)N}(\xi) &:= (Produces'_N(\xi) \wedge \neg\exists\vec{x} \in R_i) \\
&\quad \vee \Big(Produces'_N(\xi) \wedge PassThrough'_{y,M} \wedge \exists\vec{x} \in R_i\Big) \\
&\quad \bigvee_{x_j \in \vec{x}} \Big(PassThrough'_{x_j,M} \wedge First^j_{R_i}(\xi) \wedge \exists\vec{x} \in R_i\Big) \\
&\quad \bigvee_{z \in \mathrm{FV}_{\mathsf{o}}(\lambda\vec{x}.\lambda\vec{y}.M)} \Big(PassThrough'_{z,M} \wedge \xi = z \wedge \exists\vec{x} \in R_i\Big)
\end{aligned}
$$

where in the last definition, $\mathrm{FV}_{\mathsf{o}}(\lambda\vec{x}.\,\lambda\vec{y}.\,M)$ denotes the set of free variables of $\lambda\vec{x}.\,\lambda y.\,M$ of type o and the formula $First^j_{R_i}$ is a first-order formula stating that ξ is the j^{th} component of the first tuple in R_i.

Lemma 5.7 *Let $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ be a $TLI^{=}_0$ or $MLI^{=}_0$ term in canonical form, t be a subterm of Q of type τ, t' be a subterm of Q of type o, $\overline{r_1}, \ldots, \overline{r_l}$ be a legal input for Q with tuples appearing in the order given by the $Precedes_i$ predicates, $\mathrm{FV}_{\mathsf{o}}(t)$ be the set of free variables of t of type o, and ρ be a substitution that assigns constants to all variables in $\mathrm{FV}_{\mathsf{o}}(t)$. Then the following are true:*

1. *For each free variable z of t of type τ, $PassThrough_{z,t}$ defines a first-order formula with free variables in $\mathrm{FV}_{\mathsf{o}}(t)$.*
2. *$Produces_t(\xi_1, \ldots, \xi_k)$ defines a first-order formula with free variables in $\mathrm{FV}_{\mathsf{o}}(t) \cup \{\xi_1, \ldots, \xi_k\}$.*
3. *For each free variable z of t' of type o, $PassThrough'_{z,t'}$ defines a first-order formula with no free variables.*
4. *$Produces_{t'}(\xi)$ defines a first-order formula with free variables in $\mathrm{FV}_{\mathsf{o}}(t') \cup \{\xi\}$.*
5. *The term $t\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]\,\rho$ reduces to a term of the form*

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

where $m \geq 0$ and y is some free variable of t of type τ, and we have

(a) $r_1, \ldots, r_l \vdash_\rho PassThrough_{z,t}$ *iff* $z = y$,

(b) $r_1, \ldots, r_l \vdash_\rho Produces_t(\xi_1, \ldots, \xi_k)$ *iff* $\exists\, 1 \leq i \leq m\ \forall\, 1 \leq j \leq k\ \ \xi_j = o_{i,j}$,

where $r_1, \ldots, r_l \vdash_\rho \phi$ *means that the structure* $(r_1, \ldots, r_l)$ *satisfies the formula* ϕ *under valuation* ρ.

6. *The term* $t'[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$ *reduces to either a single constant* o_j *or some free variable* y *of* t' *of type* o. *In the first case, we have*

   (a) $r_1, \ldots, r_l \not\vdash PassThrough'_{z,t'}$ *for any* z,

   (b) $r_1, \ldots, r_l \vdash (Produces_{t'}(\xi) \iff \xi = o_j)$.

   In the second case, we have

   (a) $r_1, \ldots, r_l \vdash PassThrough'_{z,t'}$ *iff* $z = y$,

   (b) $r_1, \ldots, r_l \vdash (Produces'_{t'}(\xi) \iff \xi = y)$.

Proof: By structural induction over the subterms of Q and, for subterms $R_i\,(\lambda\vec{x}.\,\lambda y.\,M)\,N$, over the number of tuples in r_i. □

With this lemma, Theorem 5.1 can now easily be proved: it suffices to observe that the output of a query term $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ is described by the formula $Produces_{Q'}$.

5.3 Evaluating $\mathrm{TLI}_1^=$ and $\mathrm{MLI}_1^=$ Terms

Overview of Evaluation: We show next that $\mathrm{TLI}_1^=$ and $\mathrm{MLI}_1^=$ queries can be evaluated in time polynomial in the size of the input relations. The main idea here is to evaluate such terms not *syntactically* (i.e., by reduction), but rather *semantically*, by computing their values in an appropriate model.

For example, the prototypical iteration in a $\mathrm{TLI}_1^=/\mathrm{MLI}_1^=$ query term looks like

$$R_i\,(\lambda\vec{x}.\,\lambda f.\,Step)\,Init,$$

where R_i denotes an input relation, *Init* is some order 1 term denoting the initial value of the "accumulator", and *Step* is a term that takes an order 1 "accumulator" value f and produces a new one. It is easy to see that evaluating this iteration by means of β- and Eq-reduction alone may require exponential time, since the size of the λ-term bound to f may double at each stage. However, if the value in the "accumulator" is represented *extensionally*, i.e., as a table describing its input-output behavior, a fixed bound can be put on its size, and this enables the evaluation to be carried out in polynomial time. Suppose, for example, that the accumulator variable f is of type $\mathsf{o} \to \mathsf{o}$, i.e., denotes a function from constants to constants. If $\{o_1, \ldots, o_N\}$ is the set of constants appearing in a particular input, then for the purposes of evaluating the query on that input a table of N pairs $\{(o_i, o_j) \mid (f\,o_i) \rhd o_j,\ 1 \leq i \leq N\}$ completely characterizes the value of f. The iteration can then be carried out by computing the extension of *Init* first, storing it in an accumulator, and then evaluating *Step* once for each tuple in R_i, from last to first, with $\vec{x}$ bound to the current tuple and f bound to the current accumulator value. The result of each invocation of *Step* is converted to a table again and becomes the input of the next stage. Provided that *Step* and *Init* can be evaluated in polynomial time, the whole iteration can be carried out in polynomial time.

Of course, this approach hinges very much on the fact that when evaluating a $\mathrm{TLI}_1^=/\mathrm{MLI}_1^=$ query, we do not need to produce the exact normal form of the query, but only the relation encoded by it.

Otherwise, a "semantic" evaluator would be insufficient, since the extensional representation of a λ-term does not uniquely identify its intensional form. Thus, our evaluator is not a general-purpose λ-calculus evaluator, but rather a special one for terms obeying the input-output conventions set out in Section 3.2.

Query Term Structure: We now give a formal description of the model and semantics used by the evaluator. Let us fix a particular query $(Q\,\overline{r_1}\ldots\overline{r_l})$, where $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ is a $\mathrm{TLI}_1^=/\mathrm{MLI}_1^=$ query term in γ-canonical form for some fixed canonical typing γ and $\overline{r_1},\ldots,\overline{r_l}$ are encodings of input relations of the proper arities. Let $D = \{o_1,\ldots,o_N\}$ be the active domain of the query, i.e., the set of constants appearing in $\overline{r_1},\ldots,\overline{r_l}$ and Q, let $k_1,\ldots,k_l$ be the arities of the input relations and k be the arity of the output relation. Whenever we refer to the type of a subterm t of Q in the following, we mean the canonical type of t under γ. In general, we adopt the "Church viewpoint" in the discussion below, where each term comes with a fixed type specified by type annotations on its free and bound variables. As mentioned earlier, we may assume that Q is `let`-free, by reducing all subterms $\texttt{let}\ x = M\ \texttt{in}\ N$ to $N[x := M]$ and allowing a different type annotation on every occurrence of R_i in Q'.

The evaluation of the query consists of determining the relation encoded by the normal form of $\lambda c.\,\lambda n.\,Q'[R_1 := \overline{r_1},\ldots,R_l := \overline{r_l}]$. Since the variables c and n are λ-bound at the outermost level, they act as constants as far as the evaluation is concerned, with c behaving like a "cons" operator and n behaving like the empty list "nil". In the subsequent discussion, we will therefore consider the symbols c and n, along with Eq and $o_1,\ldots,o_N$, as constants, and if we speak of the free variables of some term t, we mean the free variables of t *except* c and n.

Semantic Evaluation: The evaluation algorithm works by manipulating values of λ-terms in a suitable model rather than the terms themselves. In this model, the types $\mathtt{o}$ and τ are base types whose domains are D and the set of k-ary relations over D, respectively, and the symbols $o_1,\ldots,o_N$, Eq, c, and n denote constant values and functions over these domains. The value of a typed term t is given by a function $[\![t]\!]$, which is defined below. (Given interpretations of the base types and constants, the meaning of terms is defined in a standard way, see e.g. [21].)

Let $\mathcal{T}$ be the set of types over base types $\mathtt{o}$ and τ, i.e., the set of types given by the grammar $\mathcal{T} := \mathtt{o} \mid \tau \mid \mathcal{T} \to \mathcal{T}$. For each type $\gamma \in \mathcal{T}$, we define the *domain* of γ, denoted by $[\![\gamma]\!]$, as follows:

1. $[\![\mathtt{o}]\!] := D$
2. $[\![\tau]\!] :=$ the set of k-ary relations over D,
3. $[\![\alpha \to \beta]\!] :=$ the set of all functions from $[\![\alpha]\!]$ to $[\![\beta]\!]$.

Let Λ be the set of simply typed, type-annotated λ-terms built from variables and constants $o_1,\ldots,o_N, Eq, c, n$, i.e., the set of terms given by the grammar

$$\Lambda := o_i : \mathtt{o} \mid Eq : \mathtt{o} \to \mathtt{o} \to \tau \to \tau \to \tau \mid c : \overbrace{\mathtt{o} \to \cdots \to \mathtt{o}}^{k} \to \tau \to \tau \mid n : \tau \mid x : \mathcal{T} \mid \lambda x : \mathcal{T}.\,\Lambda \mid \Lambda\Lambda$$

subject to the condition of well-typedness. For a term $t \in \Lambda$, an *environment* e for t is a function that maps each free variable of t to an element of the domain of its type, i.e., that maps each $x \in \mathrm{FV}(t)$ to an element of $[\![\mathrm{type}(x)]\!]$. A pair (t, e) is called a *closure*. If t does not contain free variables, we write its closure as $(t, \emptyset)$.

The *value* function $[\![t, e]\!]$ maps closures (t, e) to elements of $[\![\mathrm{type}(t)]\!]$. In defining it, we use an informal λ-notation to express functions and function application. This notation should not

be confused with the formal language Λ—it is merely a metalinguistic convenience for expressing functions concisely.

1. $[\![o_i : \mathsf{o}, \emptyset]\!] := o_i$, i.e., constants evaluate to themselves. More precisely, the syntactic expression $o_i : \mathsf{o} \in \Lambda$ evaluates to the semantic object $o_i \in [\![\mathsf{o}]\!]$, but in a slight abuse of notation, we use the same symbol for both.

2. $[\![n : \tau, \emptyset]\!] := \emptyset$, i.e., the value of n is the empty k-ary relation over D.

3. $[\![Eq : \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau, \emptyset]\!] := \lambda x.\, \lambda y.\, \lambda r.\, \lambda s.\, \textbf{if } x = y \textbf{ then } r \textbf{ else } s$, i.e., the value of Eq is the function that takes two constants $x, y \in [\![\mathsf{o}]\!]$ and two relations $r, s \in [\![\tau]\!]$ and returns r if $x = y$ and s otherwise.

4. $[\![c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau, \emptyset]\!] := \lambda x_1 \ldots \lambda x_k.\, \lambda r.\, r \cup \{(x_1, \ldots, x_k)\}$, i.e., the value of c is the function that takes k constants $x_1, \ldots, x_k \in [\![\mathsf{o}]\!]$ and a k-ary relation $r \in [\![\tau]\!]$ and returns r with the tuple $(x_1, \ldots, x_k)$ added to it.

5. $[\![x : \alpha, e]\!] := e\,(x)$, i.e., the value of a variable is whatever the environment assigns to it.

6. $[\![M\,N, e]\!] := [\![M, e]\!]\,([\![N, e]\!])$, i.e., the value of an application $M\,N$ is the result of applying the value of (M, e) to the value of (N, e).

7. $[\![\lambda x : \alpha.\, M, e]\!] := \lambda y.\, [\![M, e \cup \{x := y\}]\!]$, i.e., the value of an abstraction $\lambda x : \alpha.\, M$ is a function that maps a value $y \in [\![\alpha]\!]$ to the value of M in an environment binding x to y.

It is a standard exercise in denotational semantics to show that $[\![.]\!]$ respects reduction:

Lemma 5.8 *Let M and N be terms in Λ such that $M \rhd N$ and let e be an environment for M (and therefore for N). Then $[\![M, e]\!] = [\![N, e]\!]$.*

Proof: See, e.g., [21, Theorem 2.10]. □

The next lemma says that $[\![.]\!]$ is the "right" valuation map for our purpose:

Lemma 5.9 *Let r be the relation encoded with duplicates by the normal form of $(Q\,\overline{r_1} \ldots \overline{r_l})$. Then $r = [\![Q'\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}], \emptyset]\!]$.*

Proof: Note first that $Q'\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$ with the canonical typing γ we fixed above is a closed term of Λ, so its value is well-defined.

Let t be the normal form of $Q'\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$. Then $\lambda c.\, \lambda n.\, t$ is an encoding, with duplicates, of r and t is 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 the constants $o_{i,j}$ are drawn from the set $\{o_1, \ldots, o_N\}$. According to Lemma 5.8 and the definition of $[\![.]\!]$, we have

$$
\begin{array}{rcl}
[\![Q'\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}], \emptyset]\!] & = & [\![t, \emptyset]\!] \\
& = & \{(o_{1,1}, \ldots, o_{1,k})\} \cup \ldots \cup \{(o_{m,1}, \ldots, o_{m,k})\} \cup \{\} \\
& = & \{(o_{1,1}, \ldots, o_{1,k}), \ldots, (o_{m,1}, \ldots, o_{m,k})\} \\
& = & r
\end{array}
$$

□

Finally, we have to justify that manipulating values of terms is faster than manipulating the terms themselves. Our intent is to prove that the value of an arbitrary order 1 term $t \in \Lambda$ can be represented by a table of polynomial size (w.r.t. the size of D). This is not obvious because, $[\![\tau]\!]$:= the set of k-ary relations over D, so a function on τ has a domain exponential in the size of D. Also, since values are defined for closures and not for terms, we first have to make precise what we consider as a potential value of a Λ-term.

Definition 5.10 *Let $\alpha \in \mathcal{T}$ be some type and f be an element of $[\![\alpha]\!]$. f is called* Λ-definable *at closure level 0 if there exists a closed term $t \in \Lambda$ such that $f = [\![t, \emptyset]\!]$. f is called Λ-definable at closure level $i > 0$ if there exists a term $t \in \Lambda$ and an environment e for t such that all values assigned by e are Λ-definable at a closure level $< i$ and $f = [\![t, e]\!]$. f is called Λ-definable if it is Λ-definable at some closure level.*

Lemma 5.11 *$f \in [\![\alpha]\!]$ is Λ-definable iff it is Λ-definable at closure level 0.*

Proof: Assume that $f \in [\![\alpha]\!]$ is Λ-definable at closure level $i > 0$. Then there exist a term $t \in \Lambda$ and an environment e for t such that $f = [\![t, e]\!]$ and all values assigned by e are Λ-definable at closure levels $< i$. Assume inductively that the claim is true for Λ-definability at closure levels up to $i - 1$. Then all values assigned by e are Λ-definable at closure level 0 and it follows that e can be written as $\{x_1 := [\![t_1, \emptyset]\!], \ldots, x_m := [\![t_m, \emptyset]\!]\}$, where $x_1, \ldots, x_m$ are the free variables of t and $t_1, \ldots, t_m$ are suitable closed terms. But then

$$\begin{aligned} f &= [\![t, e]\!] \\ &= [\![\lambda x_1 \ldots \lambda x_m . t, \emptyset]\!] ([\![t_1, \emptyset]\!], \ldots, [\![t_m, \emptyset]\!]) \\ &= [\![(\lambda x_1 \ldots \lambda x_m . t)\, t_1 \ldots t_m, \emptyset]\!] \\ &= [\![t\,[x_1 := t_1, \ldots, x_m := t_m], \emptyset]\!] \end{aligned}$$

whence f is Λ-definable at closure level 0. □

As it turns out, comparatively "few" functions are Λ-definable in our model. This is again due to the fact that Λ-terms cannot distinguish between different inputs of type τ, and it provides the basis for a space-efficient representation of Λ-definable values.

Lemma 5.12 *Let $\alpha \in \mathcal{T}$ be a type of the form $\overbrace{\tau \to \cdots \to \tau}^{m} \to \tau \to \beta$, where $\beta \in \{\mathsf{o}, \tau\}$, and let $f \in [\![\alpha]\!]$ be Λ-definable. If $\beta = \mathsf{o}$, then f is constant. If $\beta = \tau$, then either f is constant or there exists an $i \in \{1, \ldots, m\}$ such that for all $(y_1, \ldots, y_m) \in [\![\tau]\!]^m$, $f(y_1, \ldots, y_m)$ is given by $f(\emptyset, \ldots, \emptyset) \cup y_i$.*

Proof: Let $t \in \Lambda$ be a term in closed normal form such that $f = [\![t, \emptyset]\!]$ (such a t can be found because of Lemmas 5.11 and 5.8).

If $\beta = \mathsf{o}$, then t is a closed normal term of type $\tau \to \cdots \to \tau \to \mathsf{o}$, and it is easy to see that t must be of the form

$$\lambda x_1 \ldots \lambda x_m . o_i,$$

where $o_i \in D$. It follows that $f = [\![t, \emptyset]\!]$ is a constant function with value o_i.

If $\beta = \tau$, then t is a closed normal term of type $\tau \to \cdots \to \tau \to \tau$, and it is easy to see that it must be of the form

$$\begin{array}{l}\lambda x_1 \ldots \lambda x_m . \\ \quad (c\, o_{1,1} \ldots o_{1,k} \\ \quad\quad (c\, o_{2,1} \ldots o_{2,k} \\ \quad\quad\quad \ldots \\ \quad\quad\quad\quad (c\, o_{l,1} \ldots o_{l,k} \\ \quad\quad\quad\quad\quad x) \ldots))\end{array}$$

where $l \geq 0$, $o_{i,j} \in D$, and $x \in \{x_1, \ldots, x_m, n\}$. If $x = n$, then f is again a constant function with value $r := \{(o_{1,1}, \ldots, o_{1,k}), \ldots, (o_{l,1}, \ldots, o_{l,k})\} = f(\emptyset, \ldots, \emptyset)$. If $x = x_i$ for some i, then for all $(y_1, \ldots, y_m) \in [\![\tau]\!]^m$, $f(y_1, \ldots, y_m) = r \cup y_i = f(\emptyset, \ldots, \emptyset) \cup y_i$. □

Lemma 5.13 *Let $\alpha \in \mathcal{T}$ be a type of the form $\beta_1 \to \cdots \to \beta_m \to \beta_{m+1}$, where each $\beta_i \in \{\mathsf{o}, \tau\}$, and let $f \in [\![\alpha]\!]$ be Λ-definable. Define the* reduced domain *of f as*

$$\mathrm{rdom}(f) := \{(y_1, \ldots, y_m) \in [\![\beta_1]\!] \times \cdots \times [\![\beta_m]\!] \mid y_i \in \{\emptyset, D^k\} \text{ if } \beta_i = \tau\}$$

and the reduced graph *of f as*

$$\mathrm{rgraph}(f) := \{(y_1, \ldots, y_m, y_{m+1}) \mid (y_1, \ldots, y_m) \in \mathrm{rdom}(f),\ y_{m+1} = f(y_1, \ldots, y_m)\}$$

Then (a) $\mathrm{rgraph}(f)$ *can be stored in space $O(N^{m+k})$ and (b) there exists a procedure Apply that, given* $\mathrm{rgraph}(f)$ *and $(y_1, \ldots, y_m) \in [\![\beta_1]\!] \times \cdots \times [\![\beta_m]\!]$, determines $f(y_1, \ldots, y_m)$ in time $O(N^{m+k})$.*

Proof: Clearly, $\mathrm{rgraph}(f)$ contains at most $(\max(2, N))^m$ tuples. The components of these tuples are either constants from D or k-ary relations over D, so each component fits into space $O(N^k)$. Thus, the whole table can be stored in space $O(N^{m+k})$.

Let $\{i_1, \ldots, i_p\}$ be the set of those $i \in \{1, \ldots, m\}$ for which $\beta_i = \tau$, and let $\{j_1, \ldots, j_q\}$ be the remainder of $\{1, \ldots, m\}$. Fix an input $\vec{y} = (y_1, \ldots, y_m) \in [\![\beta_1]\!] \times \cdots \times [\![\beta_m]\!]$ and define m-tuples $\vec{z_0}$ and $\vec{z_i}$, $i \in \{i_1, \ldots, i_p\}$, of values as follows. $\vec{z_0}$ is the m-tuple that agrees with $\vec{y}$ in positions $\{j_1, \ldots, j_q\}$ and has $\emptyset$ in positions $\{i_1, \ldots, i_p\}$. For $i \in \{i_1, \ldots, i_p\}$, $\vec{z_i}$ is the m-tuple that agrees with $\vec{z_0}$ everywhere, except that position i contains D^k. Note that $\vec{z_0}$ and the $\vec{z_i}$ are elements of $\mathrm{rdom}(f)$, so $f(\vec{z_0})$ and $f(\vec{z_i})$ can be found in $\mathrm{rgraph}(f)$.
Claim: We show that either $f(\vec{z_0}) = f(\vec{z_i})$ for all $i \in \{i_1, \ldots, i_p\}$, and in this case $f(\vec{y}) = f(\vec{z_0})$ as well, or $f(\vec{z_0}) \neq f(\vec{z_i})$ for exactly one i, and in this case $f(\vec{y}) = f(\vec{z_0}) \cup y_i$.

To see why this is correct, pick a closed term $t = \lambda x_1 \ldots \lambda x_m . t' \in \Lambda$ such that $f = [\![t, \emptyset]\!]$ and consider $f' := [\![\lambda x_{i_1} \ldots \lambda x_{i_p} . t'[x_{j_1} := y_{j_1}, \ldots, x_{j_q} := y_{j_q}], \emptyset]\!] \in [\![\tau \to \cdots \to \tau \to \beta_{m+1}]\!]$. (Note that $y_{j_1}, \ldots, y_{j_q}$ are constants from D, so it makes sense to use them as part of a Λ-term.) We have $f(\vec{y}) = f'(y_{i_1}, \ldots, y_{i_p})$, $f(\vec{z_0}) = f'(\emptyset, \ldots, \emptyset)$, and $f(\vec{z_{i_j}}) = f'(\emptyset, \ldots, \emptyset, D^k, \emptyset, \ldots, \emptyset)$, where the D^k occurs in the j^{th} argument position. Lemma 5.12 implies that f' is either constant, in which case $f(\vec{z_0}) = f(\vec{z_i}) = f(\vec{y})$ for all i, or f' adds a constant relation to one of its inputs, in which case $f(\vec{z_0}) \neq f(\vec{z_i})$ for exactly one i and $f(\vec{y}) = f(\vec{z_0}) \cup y_i$.

To summarize, *Apply* can be defined as follows:

1. Form $\vec{z_0}$ and the $\vec{z_i}$ as described above.

2. Lookup $f(\vec{z_0})$ in $\mathrm{rgraph}(f)$.

3. For each i, lookup $f(\vec{z_i})$ and compare it to $f(\vec{z_0})$. If they differ, return $f(\vec{z_0}) \cup y_i$.

4. If no differences were found, return $f(\vec{z_0})$.

These steps can be performed during a single sweep over $\mathrm{rgraph}(f)$, using time $O(N^{m+k})$. □

Eval and Apply: Every order 1 type $\alpha \in \mathcal{T}$ can be written as $\beta_1 \to \cdots \to \beta_m \to \beta_{m+1}$ with $\beta_i \in \{\mathtt{o}, \tau\}$, so what we have shown is that any Λ-definable function of order 1 (and 0, of course) can be represented by a polynomial-size table. In the previous Lemma we also described *Apply* on "complete" inputs. It is also worth pointing out that even though *Apply* was defined for "complete" inputs $(y_1, \ldots, y_m)$ only, it can easily be generalized to "partial" inputs $(y_1, \ldots, y_i)$ with $i < m$, in which case it returns the reduced graph of the function $f' := \lambda x_{i+1} \ldots \lambda x_m . f\, y_1 \ldots y_i\, x_{i+1} \ldots x_m$. This is done by simply looping over all tuples $(y_{i+1}, \ldots, y_m) \in \mathrm{rdom}(f')$ and using *Apply* to compute $f'(y_{i+1}, \ldots, y_m) = f(y_1, \ldots, y_m)$ for each. With suitable data structures, this can still be done in time $O(N^{m+k})$.

Our next step is the definition of a procedure $\mathit{Eval}(t, e)$ that takes a Λ-term t of order ≤ 1 and an environment e for t and produces $[\![t, e]\!]$ as a reduced graph. In order to be able to represent all intermediate values in polynomial space, we make the following assumptions about t and e:

1. All variables occurring in t, either free or bound, are of order ≤ 1.

2. All values assigned by e are Λ-definable and represented by reduced graphs (because of assumption (1), they must be of order ≤ 1).

To simplify the presentation of *Eval*, we also assume that e provides bindings for the constants $o_1, \ldots, o_N$, Eq, c, and n to their proper values (so that Eq, for example, is bound to the reduced graph $\{(x, y, u, v, w) \in D \times D \times \{\emptyset, D^k\} \times \{\emptyset, D^k\} \times \{\emptyset, D^k\} \mid w = u \text{ if } x = y,\ w = v \text{ if } x \neq y\}$).

The term evaluated t can take only three forms, namely $t = \lambda x . M$, $t = M_1\, M_2$, and $t = x$, where x is a symbol bound in e. The case of an application can be broken down into two subcases, namely $t = x\, M_1 \ldots M_l$, where $l \geq 1$ and x is bound in e, and $t = (\lambda x . M)\, M_1 \ldots M_l$, where $l \geq 1$ and x is chosen so that it does not occur free in $M_1, \ldots, M_l$. This gives four possibilities for t, for which $\mathit{Eval}(t, e)$ is defined as follows.

$$\mathit{Eval}(x, e) := e(x)$$

$$\mathit{Eval}(x\, M_1 \ldots M_l, e) := \mathit{Apply}(e(x), \mathit{Eval}(M_1, e), \ldots, \mathit{Eval}(M_l, e))$$

$$\begin{aligned}\mathit{Eval}(\lambda x . M, e) := \{(y_1, \ldots, y_m, y_{m+1}) \mid\ & y_1 \in D \text{ if } \mathrm{type}(x) = \mathtt{o}, \\ & y_1 \in \{\emptyset, D^k\} \text{ if } \mathrm{type}(x) = \tau, \\ & (y_2, \ldots, y_{m+1}) \in \mathit{Eval}(M, e \cup \{x \mapsto y_1\})\}\end{aligned}$$

$$\mathit{Eval}((\lambda x . M)\, M_1 \ldots M_l, e) := \mathit{Eval}(M\, M_2 \ldots M_l, e \cup \{x \mapsto \mathit{Eval}(M_1, e)\})$$

In the next two lemmas, we prove the correctness of *Eval* and analyze its running time.

Lemma 5.14 *Let $t \in \Lambda$ be a term of order ≤ 1 and e be an environment that assigns Λ-definable values of order ≤ 1 to the free variables of t. Let E be the environment that arises from e by replacing each value with its reduced graph. Then* $\mathit{Eval}(t, E) = \mathrm{rgraph}([\![t, e]\!]) = \{(y_1, \ldots, y_{m+1}) \mid (y_1, \ldots, y_m) \in \mathrm{rdom}([\![t, e]\!]),\ y_{m+1} = [\![t, e]\!](y_1, \ldots, y_m)\}$.

Proof: The proof is by induction on the size of t. We abbreviate $\mathrm{rdom}(\llbracket t, e \rrbracket)$ as δ and $(y_1, \ldots, y_m)$ as $\vec{y}$.

$$
\begin{aligned}
&\mathit{Eval}(x, E) \\
&\quad = E(x) \\
&\quad = \mathrm{rgraph}(\llbracket x, e \rrbracket) \\
&\mathit{Eval}(x\, M_1 \ldots M_l, E) \\
&\quad = \mathit{Apply}(E(x), \mathit{Eval}(M_1, E), \ldots, \mathit{Eval}(M_l, E)) \\
&\quad = \mathrm{rgraph}(e(x)(\llbracket M_1, e \rrbracket, \ldots, \llbracket M_l, e \rrbracket)) \\
&\quad = \mathrm{rgraph}(\llbracket x\, M_1 \ldots M_l, e \rrbracket) \\
&\mathit{Eval}(\lambda x.\, M, E) \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ (y_2, \ldots, y_{m+1}) \in \mathit{Eval}(M, E \cup \{x \mapsto y_1\})\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket M, e \cup \{x \mapsto y_1\} \rrbracket (y_2, \ldots, y_m)\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket \lambda x.\, M, e \rrbracket (y_1, \ldots, y_m)\} \\
&\mathit{Eval}((\lambda x.\, M)\, M_1 \ldots M_l, E) \\
&\quad = \mathit{Eval}(M\, M_2 \ldots M_l, E \cup \{x \mapsto \mathit{Eval}(M_1, E)\}) \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket M\, M_2 \ldots M_l, e \cup \{x \mapsto \llbracket M_1, e \rrbracket\} \rrbracket (y_1, \ldots, y_m)\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket \lambda x.\, M\, M_2 \ldots M_l, e \rrbracket (\llbracket M_1, e \rrbracket, y_1, \ldots, y_m)\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket (\lambda x.\, M\, M_2 \ldots M_l)\, M_1, e \rrbracket (y_1, \ldots, y_m)\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket (M\, M_2 \ldots M_l)[x := M_1], e \rrbracket (y_1, \ldots, y_m)\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket M[x := M_1]\, M_2 \ldots M_l, e \rrbracket (y_1, \ldots, y_m)\} \\
&\quad = \{(y_1, \ldots, y_{m+1}) \mid \vec{y} \in \delta,\ y_{m+1} = \llbracket (\lambda x.\, M)\, M_1 \ldots M_l, e \rrbracket (y_1, \ldots, y_m)\}
\end{aligned}
$$

□

To analyze the running time of *Eval*, we need to introduce the following quantity. For $t \in \Lambda$, let $d(t)$ be defined by

$$
\begin{aligned}
d(x) &:= 0 \quad (\text{where } x \text{ is a variable or a constant}) \\
d(M_1\, M_2) &:= \max(d(M_1), d(M_2)) \\
d(\lambda x.\, M) &:= \begin{cases} 1 + d(M) & \text{if } \mathrm{order}(x) = 0 \\ d(M) & \text{otherwise} \end{cases}
\end{aligned}
$$

Looking at the syntax tree of t, $d(t)$ counts the maximum number of order 0 abstractions along any path from the root to a leaf. This number figures in the running time of $\mathit{Eval}(t, e)$ because, whereas subterms of t are usually evaluated only once, the body of an order 0 abstraction is evaluated many times, once for every possible value of the bound variable in the reduced domain of t.

Lemma 5.15 *Let t and E be as in Lemma 5.14, let $\#(t, E)$ be the number of recursive calls to Eval resulting from the call $\mathrm{Eval}(t, E)$ (including the initial call), and let $|t|$ be the size of t. Then $\#(t, E) \leq (\max(2, N))^{d(t)}|t|$.*

Proof: The proof is by induction on $|t|$. To save some ink, we assume that $N \geq 2$.

$$
\begin{aligned}
&\#(x, E) \\
&\quad = 1 \\
&\quad = N^{d(x)}|x|
\end{aligned}
$$

$$
\begin{aligned}
&\#\,(x\,M_1\ldots M_l, E)\\
&\quad = 1 + \textstyle\sum_{i=1}^{l} \#\,(M_i, E)\\
&\quad \le 1 + \textstyle\sum_{i=1}^{l} N^{d(M_i)}|M_i|\\
&\quad \le N^{\max\{d(M_i)\}}\,(1 + \textstyle\sum_{i=1}^{l}|M_i|)\\
&\quad = N^{d(xM_1\ldots M_l)}|x\,M_1\ldots M_l|
\end{aligned}
$$

$$
\begin{aligned}
&\#\,(\lambda x.\,M, E)\\
&\quad = 1 + \begin{cases} \sum_{y\in D} \#\,(M, E\cup\{x\mapsto y\}) & \text{if } \mathrm{type}\,(x) = \mathsf{o}\\ \#\,(M, E\cup\{x\mapsto \emptyset\}) + \#\,(M, E\cup\{x\mapsto D^k\}) & \text{if } \mathrm{type}\,(x) = \tau \end{cases}\\
&\quad \le 1 + N\,N^{d(M)}|M|\\
&\quad \le N^{d(M)+1}\,(|M|+1)\\
&\quad = N^{d(\lambda x.M)}|\lambda x.\,M|
\end{aligned}
$$

$$
\begin{aligned}
&\#\,((\lambda x.\,M)\,M_1\ldots M_l, E)\\
&\quad = 1 + \#\,(M_1, E) + \#\,(M\,M_2\ldots M_l, E\cup\{x\mapsto \mathit{Eval}\,(M_1, E)\})\\
&\quad \le 1 + N^{d(M_1)}|M_1| + N^{\max\{d(M),d(M_2),\ldots,d(M_l)\}}\,(|M| + \textstyle\sum_{i=2}^{l}|M_i|)\\
&\quad \le N^{\max\{d(M),d(M_1),\ldots,d(M_l)\}}\,(1 + |M| + \textstyle\sum_{i=1}^{l}|M_i|)\\
&\quad \le N^{d((\lambda x.M)M_1\ldots M_l)}|(\lambda x.\,M)\,M_1\ldots M_l|
\end{aligned}
$$

□

Having analyzed the correctness and running time of the semantic evaluator, we have to bring the query (applied to the input) to the proper form to use this evaluator by reducing the "top-level" redexes.

Lemma 5.16 *Let $Q = \lambda R_1\ldots\lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ be a $TLI_1^=$ or $MLI_1^=$ query term in canonical form and $\overline{r_1},\ldots,\overline{r_l}$ be encodings of relations of the proper arities for Q. Let t be the Λ-term that arises from Q' by replacing every subterm of the form $R_i\,M\,N$, where $i\in\{1,\ldots,l\}$, by the term*

$$
\begin{aligned}
&(M\,o_{1,1}\ldots o_{1,k}\\
&\quad (M\,o_{2,1}\ldots o_{2,k}\\
&\qquad \ldots\\
&\qquad\quad (M\,o_{m,1}\ldots o_{m,k}\,N))\ldots),
\end{aligned}
$$

where the $o_{i,j}$ are determined by

$$
\begin{aligned}
&\overline{r_i} = \lambda c.\,\lambda n.\\
&\qquad (c\,o_{1,1}\,o_{1,2}\ldots o_{1,k}\\
&\qquad\quad (c\,o_{2,1}\,o_{2,2}\ldots o_{2,k}\\
&\qquad\qquad \ldots\\
&\qquad\qquad\quad (c\,o_{m,1}\,o_{m,2}\ldots o_{m,k}\,n))\ldots).
\end{aligned}
$$

That is, t is the result of reducing the "top-level" redexes in $(Q\,\overline{r_1}\ldots\overline{r_l})$. Let $m := \max\{|\overline{r_1}|,\ldots,|\overline{r_l}|\}$. Then $|t|\le m^{|Q'|}$ and $d\,(t) = d\,(Q')$.

Proof: For any subterm S of Q', let $\widetilde{S}$ denote the result of carrying out the replacements described above (thus, $t = \widetilde{Q'}$). We show by induction on $|S|$ that $|\widetilde{S}|\le m^{|S|}$.

There are three cases to consider: $S = x\,M_1 \ldots M_n$, where $n \geq 0$ and x is a constant or variable not in $\{R_1, \ldots, R_l\}$, $S = R_i\,M_1 \ldots M_n$, where we may assume that $n \geq 2$ due to canonicity, and $S = (\lambda x.\,M)\,M_1 \ldots M_n$, where $n \geq 0$. For the first and third cases, it is straightforward to show that $|\widetilde{S}| \leq m^{|S|}$. For the second case, we have

$$\begin{aligned}
|\widetilde{S}| &= |(\widetilde{M_1}\vec{o_1}\,(\widetilde{M_1}\vec{o_2} \ldots (\widetilde{M_1}\vec{o_j}\,\widetilde{M_2})) \ldots)\,\widetilde{M_3} \ldots \widetilde{M_n}| \\
&\quad (\text{where } \vec{o_1}, \ldots, \vec{o_j} \text{ are taken from } \overline{r_i}) \\
&\leq |\overline{r_i}||\widetilde{M_1}| + \sum_{i=2}^{n} |\widetilde{M_i}| \\
&\leq m\,m^{|M_1|} + \sum_{i=2}^{n} m^{|M_i|} \\
&\leq m^{1+\sum_{i=1}^{n}|M_i|} \\
&= m^{|S|}.
\end{aligned}$$

To show that $d\,(t) = d\,(Q')$, observe that

$$\begin{aligned}
d\,(R_i\,M\,N) &= \max\{d\,(M), d\,(N)\} \\
&= d\,(M\,\vec{o_1}\,(M\,\vec{o_2} \ldots (M\,\vec{o_j}\,N)) \ldots).
\end{aligned}$$

It is easy to see that this implies $d\,(t) = d\,(Q')$. □

Lemma 5.17 *Database queries defined by terms in $TLI_1^=$ and $MLI_1^=$ can be evaluated in PTIME.*

Proof: Let a canonical $TLI_1^=$ or $MLI_1^=$ query term $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ and inputs $\overline{r_1}, \ldots, \overline{r_l}$ be given. Form t as described in the previous lemma (this can clearly be done in polynomial time) and compute the active domain D of the query (which is needed by *Eval* to compute the extents of the types o and τ). Observe that t is a closed Λ-term and that Lemma 5.5 implies that t does not contain variables of order ≥ 2, so that $(t, \emptyset)$ satisfies the assumptions made in the definition of *Eval*. By virtue of Lemmas 5.9 and 5.14–5.16, $Eval\,(t, \emptyset)$ produces the output of the query in time polynomial in the size of $\overline{r_1}, \ldots, \overline{r_l}$. □

Thus, the proof of Theorem 5.2 is complete.

6 ML Type Reconstruction for Fixed Order Functionalities

In this section, we extend previous results on the complexity of ML typing by showing that ML type reconstruction for terms of bounded order is NP-hard. (This also corrects an erroneous claim of PTIME-membership that we made in [26].)

ML type reconstruction in general is EXPTIME-complete, but the known proofs [31, 32] use terms of unbounded order. We do not know whether EXPTIME completeness holds also for terms of bounded order. Our result is the following:

Theorem 6.1 *For each fixed $k \geq 4$, type reconstruction in order k core-ML is NP-hard in the program size. Type reconstruction for each $MLI_i^=$, $i \geq 1$, is NP-hard in the query term size.*

Proof: The proof is by reduction from CNF-satisfiability. Let a propositional formula $\Phi = C_1 \wedge \ldots \wedge C_m$ be given, where each clause C_i is a disjunction of a set of literals chosen from the set

$\{v_1, \overline{v_1}, v_2, \overline{v_2}, \ldots, v_n, \overline{v_n}\}$. We construct a term E_Φ whose length is polynomial in the length of Φ such that E_Φ is *not* typable in order 4 core-ML if and only if Φ is satisfiable.

The reduction works as follows. Imagine the variables $v_1, \ldots, v_n$ arranged in a truth table TAB. The column associated with variable v_n looks like $F, T, F, T, \ldots$, the column associated with v_{n-1} looks like $F, F, T, T, \ldots$ and so forth. If we interpret T as 1 and F as 0, then the value $b_{i,j}$ in the i^{th} row and j^{th} column of TAB is the j^{th} bit in the binary expansion of i (where the bits are numbered $1, 2, \ldots, n$ going from left to right).

For each column j of the truth table, we construct an ML term V_j whose principal type is an exponentially long order 1 type of the form

$$\begin{aligned}\phi_j = \alpha^j &\to \beta^j_{1,1} \to \gamma^j_{1,1} \to \beta^j_{1,2} \to \gamma^j_{1,2} \to \cdots \to \beta^j_{1,m} \to \gamma^j_{1,m} \\ &\to \beta^j_{2,1} \to \gamma^j_{2,1} \to \beta^j_{2,2} \to \gamma^j_{2,2} \to \cdots \to \beta^j_{2,m} \to \gamma^j_{2,m} \\ &\cdots \\ &\to \beta^j_{2^n,1} \to \gamma^j_{2^n,1} \to \beta^j_{2^n,2} \to \gamma^j_{2^n,2} \to \cdots \to \beta^j_{2^n,m} \to \gamma^j_{2^n,m} \\ &\to \alpha^j\end{aligned}$$

In this type, the $\beta^j_{k,l}$ and $\gamma^j_{k,l}$ denote pairwise distinct type variables, *except* that for certain k and l, $\beta^j_{k,l}$ and $\gamma^j_{k,l}$ are unified, that is, they denote the same type variable. (Think of a pair $(\beta^j_{k,l}, \gamma^j_{k,l})$ as a bit that is "off" if $\beta^j_{k,l}$ and $\gamma^j_{k,l}$ are distinct variables and "on" if they are identical.) The choice which pairs $(\beta^j_{k,l}, \gamma^j_{k,l})$ to unify is made as follows. The fragment

$$\cdots \to \beta^j_{k,1} \to \gamma^j_{k,1} \to \beta^j_{k,2} \to \gamma^j_{k,2} \to \cdots \to \beta^j_{k,m} \to \gamma^j_{k,m} \to \cdots$$

(which we will call the k^{th} *segment* of ϕ_j for lack of a better term) corresponds to the truth value in row k of column j, that is $b_{k,j}$. A pair $(\beta^j_{k,l}, \gamma^j_{k,l})$ in this segment is unified if and only if assigning value $b_{k,j}$ to v_j makes clause C_l true. Put another way, $(\beta^j_{k,l}, \gamma^j_{k,l})$ is unified iff either $b_{k,j} = T$ and v_j occurs positively in C_l or $b_{k,j} = F$ and v_j occurs negatively in C_l.

An example might be in order here. Suppose we have two variables v_1, v_2 and three clauses $C_1 = v_1 \vee v_2$, $C_2 = v_1 \vee \overline{v_2}$, $C_3 = \overline{v_2}$. The truth table for v_1 and v_2 is

v_1	v_2
F	F
F	T
T	F
T	T

and V_1 will be engineered (we haven't yet said how) to have principal type

$$\begin{aligned}\phi_1 = \alpha^1 &\to \beta^1_{1,1} \to \gamma^1_{1,1} \to \beta^1_{1,2} \to \gamma^1_{1,2} \to \beta^1_{1,3} \to \gamma^1_{1,3} \\ &\to \beta^1_{2,1} \to \gamma^1_{2,1} \to \beta^1_{2,2} \to \gamma^1_{2,2} \to \beta^1_{2,3} \to \gamma^1_{2,3} \\ &\to \beta^1_{3,1} \to \beta^1_{3,1} \to \beta^1_{3,2} \to \beta^1_{3,2} \to \beta^1_{3,3} \to \gamma^1_{3,3} \\ &\to \beta^1_{4,1} \to \beta^1_{4,1} \to \beta^1_{4,2} \to \beta^1_{4,2} \to \beta^1_{4,3} \to \gamma^1_{4,3} \\ &\to \alpha^1.\end{aligned}$$

The principal type of V_2 will be

$$\begin{aligned}\phi_2 = \alpha^2 &\to \beta^2_{1,1} \to \gamma^2_{1,1} \to \beta^2_{1,2} \to \beta^2_{1,2} \to \beta^2_{1,3} \to \beta^2_{1,3} \\ &\to \beta^2_{2,1} \to \beta^2_{2,1} \to \beta^2_{2,2} \to \gamma^2_{2,2} \to \beta^2_{2,3} \to \gamma^2_{2,3} \\ &\to \beta^2_{3,1} \to \gamma^2_{3,1} \to \beta^2_{3,2} \to \beta^2_{3,2} \to \beta^2_{3,3} \to \beta^2_{3,3} \\ &\to \beta^2_{4,1} \to \beta^2_{4,1} \to \beta^2_{4,2} \to \gamma^2_{4,2} \to \beta^2_{4,3} \to \gamma^2_{4,3} \\ &\to \alpha^2.\end{aligned}$$

Returning to the general case, we build, in addition to the terms $V_1, \ldots, V_n$, a term V_0 whose principal type ϕ_0 has the same general structure as that of $\phi_1, \ldots, \phi_n$, but with the unifications arranged slightly differently. Namely, in every segment

$$\cdots \to \beta^0_{k,1} \to \gamma^0_{k,1} \to \beta^0_{k,2} \to \gamma^0_{k,2} \to \cdots \to \beta^0_{k,m} \to \gamma^0_{k,m} \to \cdots$$

of ϕ_0, we unify $\gamma^0_{k,1}$ with $\beta^0_{k,2}$, $\gamma^0_{k,2}$ with $\beta^0_{k,3}$, and so on, up to $\gamma^0_{k,m-1}$ and $\beta^0_{k,m}$. In addition, we arrange things so that $\beta^0_{k,1}$ will *not* unify with $\gamma^0_{k,m}$, for example by making $\beta^0_{k,1}$ an arrow type of the form $\gamma^0_{k,m} \to \delta$.

Having constructed $V_0, V_1, \ldots, V_n$, we use them as part of a larger term W where the context is such that it forces the types of $V_0, V_1, \ldots, V_n$ to be unified. This causes interesting things to happen. Each V_j is assigned a common type

$$\begin{aligned}\phi = \alpha &\to \beta_{1,1} \to \gamma_{1,1} \to \beta_{1,2} \to \gamma_{1,2} \to \cdots \to \beta_{1,m} \to \gamma_{1,m} \\ &\to \beta_{2,1} \to \gamma_{2,1} \to \beta_{2,2} \to \gamma_{2,2} \to \cdots \to \beta_{2,m} \to \gamma_{2,m} \\ &\cdots \\ &\to \beta_{2^n,1} \to \gamma_{2^n,1} \to \beta_{2^n,2} \to \gamma_{2^n,2} \to \cdots \to \beta_{2^n,m} \to \gamma_{2^n,m} \\ &\to \alpha\end{aligned}$$

which is unified with each ϕ_j, $0 \le j \le n$. In particular, each $\beta_{k,l}$ is unified with all $\beta^j_{k,l}$, $0 \le j \le n$, and the same goes for $\gamma_{k,l}$. Let us investigate under which conditions a pair $(\beta_{k,l}, \gamma_{k,l})$ in ϕ is unified. Since ϕ_0 does not unify any pairs $(\beta^0_{k,l}, \gamma^0_{k,l})$, this happens if and only if there exists a $j \in \{1, \ldots, n\}$ such that $\beta^j_{k,l}$ and $\gamma^j_{k,l}$ are unified in ϕ_j, which happens if and only if assigning $b_{k,j}$ to v_j makes C_l true. In other words, $\beta_{k,l}$ and $\gamma_{k,l}$ are unified if and only if clause C_l is true under the truth assignment given by the k^{th} row of TAB. If for some k, *all* pairs $(\beta_{k,l}, \gamma_{k,l})$, $1 \le l \le m$, are unified, then the truth assignment in the k^{th} row of TAB makes Φ true and hence Φ must be satisfiable.

Note, however, the unifications imposed upon ϕ by ϕ_0. For every $k \in \{1, \ldots, 2^n\}$, ϕ_0 unifies the pairs $(\gamma^0_{k,l}, \beta^0_{k,l+1})$, $1 \le l < m$ (we still haven't said how this is arranged, but will do so shortly). As a consequence, for every $k \in \{1, \ldots, 2^n\}$, the pairs $(\gamma_{k,l}, \beta_{k,l+1})$, $1 \le l < m$, are unified in ϕ. Now *if* Φ is satisfiable, i.e., if for some k the pairs $(\beta_{k,l}, \gamma_{k,l})$ are unified for $1 \le l < m$, this results in the entire set $\{\beta_{k,1}, \gamma_{k,1}, \ldots, \beta_{k,m}, \gamma_{k,m}\}$ being unified. But ϕ_0 has arranged that $\beta^0_{k,1}$ and $\gamma^0_{k,m}$, and therefore $\beta_{k,1}$ and $\gamma_{k,m}$, are not unifiable, so the unification of $\phi_0, \phi_1, \ldots, \phi_n$ fails. Thus, if Φ is satisfiable, W cannot be typed, whereas if Φ is not satisfiable, the unification of $\phi_0, \phi_1, \ldots, \phi_n$ succeeds and W can be typed.

Now that we have described how the reduction works at the type level, we have to construct polynomial-size ML terms $V_0, V_1, \ldots, V_n$ that actually have these types. Here is where `let`-polymorphism comes in, because it allows us to write succinct terms that have very large principal types.

The first step in the construction is to show how to unify the type of two terms. Suppose we are given two terms M and N and we want to force a common type upon them that is the most general unifier of their principal types. This can be done by the following term:

$$\mathit{Unify}_{(M,N)} := \lambda z.\, K\, z\, (\lambda f.\, K\, (f\, M)\, (f\, N)),$$

where K is the combinator $\lambda x.\, \lambda y.\, x$. The type of $\mathit{Unify}_{(M,N)}$ is $\alpha \to \alpha$ independent of M and N, but in the derivation of this type, the types of M and N are unified, because the same function f is applied to both M and N. Note that the typing of $\mathit{Unify}_{M,N}$ can be carried out in order $k+3$, where k is the order of the common type assigned to M and N.

As a next step, let us see how to build terms whose principal types can serve as segments of the types ϕ_j. Consider the term

$$\mathit{Template} := \lambda z.\, \lambda x_1.\, \lambda y_1.\, \lambda x_2.\, \lambda y_2 \ldots \lambda x_m.\, \lambda y_m.\, K\, z\, w,$$

where w is a free variable. Its principal type is

$$\alpha \to \beta_1 \to \gamma_1 \to \beta_2 \to \gamma_2 \to \cdots \to \beta_m \to \gamma_m \to \alpha,$$

no matter what the type of w is, and it can be typed in order 1 core-ML. Suppose now that we want to unify certain pairs of type variables in this type, say (β_1, γ_1) and (β_3, γ_3). This is achieved by the following variant of *Template*:

$$\mathit{Template}_{(1,1),(3,3)} := \lambda z.\, \lambda x_1.\, \lambda y_1.\, \lambda x_2.\, \lambda y_2 \ldots \lambda x_m.\, \lambda y_m.\, K\, z\, (\mathit{Unify}_{(x_1,y_1)}\, (\mathit{Unify}_{(x_3,y_3)}\, z)),$$

which can be typed in order 3 core-ML (because of *Unify*) with principal type

$$\alpha \to \beta_1 \to \beta_1 \to \beta_2 \to \gamma_2 \to \beta_3 \to \beta_3 \to \cdots \to \beta_m \to \gamma_m \to \alpha.$$

As another example, in the type ϕ_0 we want segments that look like

$$\cdots \to \beta_1 \to \gamma_1 \to \gamma_1 \to \gamma_2 \to \gamma_2 \to \cdots \to \gamma_{m-1} \to \gamma_{m-1} \to \gamma_m \to \cdots$$

where β_1 and γ_m are *not* unifiable. Such a segment can be generated by another variant of *Template*,

$$\begin{aligned}\mathit{Oddpairs} := \lambda z.\, &\lambda x_1.\, \lambda y_1.\, \lambda x_2.\, \lambda y_2 \ldots \lambda x_m.\, \lambda y_m. \\ &K\, z\, (\mathit{Unify}_{(y_1,x_2)}\, (\mathit{Unify}_{(y_2,x_3)} \ldots (\mathit{Unify}_{(y_{m-1},x_m)}\, (x_1\, y_m))) \ldots)\end{aligned}$$

which is typable in order 3 core-ML with principal type

$$\alpha \to (\gamma_m \to \delta) \to \gamma_1 \to \gamma_1 \to \gamma_2 \to \gamma_2 \to \cdots \to \gamma_{m-1} \to \gamma_{m-1} \to \gamma_m \to \alpha.$$

Consider now the construction of the term V_0. We want it to have a principal type of the form

$$\begin{aligned}\phi_j = \alpha^0 &\to \beta^0_{1,1} \to \gamma^0_{1,1} \to \beta^0_{1,2} \to \gamma^0_{1,2} \to \cdots \to \beta^0_{1,m} \to \gamma^0_{1,m} \\ &\to \beta^0_{2,1} \to \gamma^0_{2,1} \to \beta^0_{2,2} \to \gamma^0_{2,2} \to \cdots \to \beta^0_{2,m} \to \gamma^0_{2,m} \\ &\cdots \\ &\to \beta^0_{2^n,1} \to \gamma^0_{2^n,1} \to \beta^0_{2^n,2} \to \gamma^0_{2^n,2} \to \cdots \to \beta^0_{2^n,m} \to \gamma^0_{2^n,m} \\ &\to \alpha^0\end{aligned}$$

where $\gamma^0_{k,l+1} = \beta^0_{k,l}$ for $1 \leq l < m$ and $\beta^0_{k,1}$ and $\gamma^0_{k,m}$ are not unifiable. This is achieved by "stringing together" many copies of *Oddpairs* using `let`:

$$
\begin{array}{l}
V_0 := \texttt{let}\ s_0 = \mathit{Oddpairs}\ \texttt{in} \\
\qquad \texttt{let}\ s_1 = \lambda z.\, s_0\,(s_0\,z)\ \texttt{in} \\
\qquad\quad \texttt{let}\ s_2 = \lambda z.\, s_1\,(s_1\,z)\ \texttt{in} \\
\qquad\qquad \cdots \\
\qquad\qquad\quad \texttt{let}\ s_n = \lambda z.\, s_{n-1}\,(s_{n-1}\,z)\ \texttt{in} \\
\qquad\qquad\qquad s_n
\end{array}
$$

Note that V_0 is of polynomial size. We prove by induction on n that s_n (and thus V_0) has principal type

$$
\begin{array}{l}
\phi^n_0 = \alpha^0 \to (\gamma^0_{1,m} \to \delta_1) \to \gamma^0_{1,1} \to \gamma^0_{1,1} \to \gamma^0_{1,2} \to \gamma^0_{1,2} \to \cdots \to \gamma^0_{1,m-1} \to \gamma^0_{1,m-1} \to \gamma^0_{1,m} \\
\qquad \to (\gamma^0_{2,m} \to \delta_2) \to \gamma^0_{2,1} \to \gamma^0_{2,1} \to \gamma^0_{2,2} \to \gamma^0_{2,2} \to \cdots \to \gamma^0_{2,m-1} \to \gamma^0_{2,m-1} \to \gamma^0_{2,m} \\
\qquad \cdots \\
\qquad \to (\gamma^0_{2^n,m} \to \delta_{2^n}) \to \gamma^0_{2^n,1} \to \gamma^0_{2^n,1} \to \gamma^0_{2^n,2} \to \gamma^0_{2^n,2} \to \cdots \to \gamma^0_{2^n,m-1} \to \gamma^0_{2^n,m-1} \to \gamma^0_{2^n,m} \\
\qquad \to \alpha^0
\end{array}
$$

In the base case $n = 0$, we have $s_0 = \mathit{Oddpairs}$, so the claim follows from the type of *Oddpairs* given above. For $n > 0$, the inductive hypothesis implies that s_{n-1} has principal type ϕ^{n-1}_0. The principal type of s_n is that of $\lambda z.\, s_{n-1}\,(s_{n-1}\,z)$ with the two occurrences of s_{n-1} typed independently. Let $\tilde{\phi}^{n-1}_0$ be a copy of ϕ^{n-1}_0 with the type variables renamed and define ψ and $\tilde{\psi}$ by $\phi^{n-1}_0 = \alpha^0 \to \psi$, $\tilde{\phi}^{n-1}_0 = \tilde{\alpha}^0 \to \tilde{\psi}$. Then the principal type of s_n is given by $\alpha^0 \to \psi\,[\alpha^0 := \tilde{\psi}\,[\tilde{\alpha}^0 := \alpha^0]]$, which, after renaming of type variables, is ϕ^n_0. Moreover, in deriving this type, the two occurrences of s_{n-1} in $\lambda z.\, s_{n-1}\,(s_{n-1}\,z)$ are typed as $\phi^{n-1}_0\,[\alpha^0 := \psi]$ and ϕ^{n-1}_0, respectively, so the derivation can be carried out in order 3 core-ML.

The construction of V_j, where $j \in \{1, \ldots, n\}$, is similar to that of V_0. Fix a variable v_j and let $\{C_{i_1}, \ldots, C_{i_p}\}$ be the set of clauses in which v_j occurs positively and $\{C_{j_1}, \ldots, C_{j_q}\}$ be the set of clause in which v_j occurs negatively. Define "segment-generating" terms T_j and F_j as follows:

$$
\begin{array}{l}
T_j := \lambda z.\, \lambda x_1.\, \lambda y_1.\, \lambda x_2.\, \lambda y_2 \ldots \lambda x_m.\, \lambda y_m. \\
\qquad K\,z\,(\mathit{Unify}_{(x_{i_1}, y_{i_1})}\,(\mathit{Unify}_{(x_{i_2}, y_{i_2})} \ldots (\mathit{Unify}_{(x_{i_p}, y_{i_p})}\,z)) \ldots) \\
F_j := \lambda z.\, \lambda x_1.\, \lambda y_1.\, \lambda x_2.\, \lambda y_2 \ldots \lambda x_m.\, \lambda y_m. \\
\qquad K\,z\,(\mathit{Unify}_{(x_{j_1}, y_{j_1})}\,(\mathit{Unify}_{(x_{j_2}, y_{j_2})} \ldots (\mathit{Unify}_{(x_{j_q}, y_{j_q})}\,z)) \ldots)
\end{array}
$$

These terms are typable in order 3 core-ML with principal types of the form

$$\alpha \to \beta_1 \to \gamma_1 \to \beta_2 \to \gamma_2 \to \cdots \to \beta_m \to \gamma_m \to \alpha,$$

where in the case of T_j, $\beta_l = \gamma_l$ iff v_j occurs positively in C_l, and in the case of F_j, $\beta_l = \gamma_l$ iff v_j occurs negatively in C_l.

The term V_j is built from 2^n copies of F_j and T_j "strung together" in a pattern that mimics the sequence of truth values in the j^{th} column of TAB. That is, a block of 2^{n-j} copies of F_j is followed by a block of 2^{n-j} copies of T_j, followed by another block of F_j's, and so on, for a total of 2^j blocks. Again, we use `let` to generate this pattern succinctly:

$$
\begin{array}{l}
V_j := \texttt{let } f_0 = F_j \texttt{ in} \\
\quad\quad \texttt{let } f_1 = \lambda z.\, f_0\,(f_0\,z) \texttt{ in} \\
\quad\quad\quad \cdots \\
\quad\quad\quad\quad \texttt{let } f_{n-j} = \lambda z.\, f_{n-j-1}\,(f_{n-j-1}\,z) \texttt{ in} \\
\quad\quad\quad\quad\quad \texttt{let } t_0 = T_j \texttt{ in} \\
\quad\quad\quad\quad\quad\quad \texttt{let } t_1 = \lambda z.\, t_0\,(t_0\,z) \texttt{ in} \\
\quad\quad\quad\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{let } t_{n-j} = \lambda z.\, t_{n-j-1}\,(t_{n-j-1}\,z) \texttt{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{let } s_1 = \lambda z.\, f_{n-j}\,(t_{n-j}\,z) \texttt{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{let } s_2 = \lambda z.\, s_1\,(s_1\,z) \texttt{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{let } s_j = \lambda z.\, s_{j-1}\,(s_{j-1}\,z) \texttt{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad s_j
\end{array}
$$

Note that the term V_j is of polynomial size. We claim that its principal type has the form

$$
\begin{array}{l}
\phi_j = \alpha^j \to \beta^j_{1,1} \to \gamma^j_{1,1} \to \beta^j_{1,2} \to \gamma^j_{1,2} \to \cdots \to \beta^j_{1,m} \to \gamma^j_{1,m} \\
\quad\quad \to \beta^j_{2,1} \to \gamma^j_{2,1} \to \beta^j_{2,2} \to \gamma^j_{2,2} \to \cdots \to \beta^j_{2,m} \to \gamma^j_{2,m} \\
\quad\quad \cdots \\
\quad\quad \to \beta^j_{2^n,1} \to \gamma^j_{2^n,1} \to \beta^j_{2^n,2} \to \gamma^j_{2^n,2} \to \cdots \to \beta^j_{2^n,m} \to \gamma^j_{2^n,m} \\
\quad\quad \to \alpha^j,
\end{array}
$$

where $\beta^j_{k,l} = \gamma^j_{k,l}$ iff either the j^{th} bit in the binary expansion of k, i.e. $b_{k,j}$, is 1 and v_j occurs positively in clause C_l or the bit is 0 and v_j occurs negatively in clause C_l.

To this end, we first show by induction on i that f_i and t_i can be typed in order 3 core-ML with principal types of the form

$$
\begin{array}{l}
\alpha^j \to \beta^j_{1,1} \to \gamma^j_{1,1} \to \beta^j_{1,2} \to \gamma^j_{1,2} \to \cdots \to \beta^j_{1,m} \to \gamma^j_{1,m} \\
\quad \to \beta^j_{2,1} \to \gamma^j_{2,1} \to \beta^j_{2,2} \to \gamma^j_{2,2} \to \cdots \to \beta^j_{2,m} \to \gamma^j_{2,m} \\
\quad \cdots \\
\quad \to \beta^j_{2^i,1} \to \gamma^j_{2^i,1} \to \beta^j_{2^i,2} \to \gamma^j_{2^i,2} \to \cdots \to \beta^j_{2^i,m} \to \gamma^j_{2^i,m} \\
\quad \to \alpha^j,
\end{array}
$$

where in the case of f_i, $\beta^j_{k,l} = \gamma^j_{k,l}$ iff v_j occurs negatively in C_l and in the case of t_i, $\beta^j_{k,l} = \gamma^j_{k,l}$ iff v_j occurs negatively in C_l. The induction is exactly analogous to the one carried out for V_0 and is not repeated here.

Let us abbreviate the principal types of f_{n-j} and t_{n-j} as $\overline{\alpha}^j \to \Omega_F \to \overline{\alpha}^j$ and $\alpha^j \to \Omega_T \to \alpha^j$, respectively. The understanding here is that Ω_F and Ω_T are metasyntactic variables (not types) that stand for strings of the form $\beta^j_{1,1} \to \gamma^j_{1,1} \to \cdots \to \beta^j_{2^{n-j},m} \to \gamma^j_{2^{n-j},m}$ where certain $\beta^j_{k,l}$ and $\gamma^j_{k,l}$ are equal as described above. We also adopt the convention that in every occurrence of Ω_F or Ω_T, the type variables are renamed to be distinct from those in any other occurrence of Ω_F or Ω_T.

Under these conventions, the principal type of $s_1 = \lambda z.\, f_{n-j}\,(t_{n-j}\,z)$ can be written as $\alpha^j \to \Omega_F \to \Omega_T \to \alpha^j$. By induction on i, we can show that the principal type of s_i is $\alpha^j \to \Omega_F \to$

$\Omega_T \to \cdots \to \Omega_F \to \Omega_T \to \alpha^j$, where there are 2^i occurrences of Ω_F and Ω_T altogether. Thus, the principal type of s_j (and therefore V_j) has the form $\alpha^j \to \Omega_F \to \Omega_T \to \cdots \to \Omega_F \to \Omega_T \to \alpha^j$ with 2^j occurrences of Ω_F or Ω_T altogether, each of which contributes 2^{n-j} segments to the type. After a suitable renumbering of type variables, the principal type of V_j can therefore be written as

$$\begin{aligned}
\alpha^j &\to \beta^j_{1,1} \to \gamma^j_{1,1} \to \beta^j_{1,2} \to \gamma^j_{1,2} \to \cdots \to \beta^j_{1,m} \to \gamma^j_{1,m} \\
&\to \beta^j_{2,1} \to \gamma^j_{2,1} \to \beta^j_{2,2} \to \gamma^j_{2,2} \to \cdots \to \beta^j_{2,m} \to \gamma^j_{2,m} \\
&\cdots \\
&\to \beta^j_{2^n,1} \to \gamma^j_{2^n,1} \to \beta^j_{2^n,2} \to \gamma^j_{2^n,2} \to \cdots \to \beta^j_{2^n,m} \to \gamma^j_{2^n,m} \\
&\to \alpha^j,
\end{aligned}$$

where the first 2^{n-j} segments are contributed by a block Ω_F, the next 2^{n-j} segments are contributed by a block Ω_T, and so forth. Consider a pair $(\beta^j_{k,l}, \gamma^j_{k,l})$. It is easy to see that this pair was contributed by a block Ω_F if the j^{th} bit (from the left) in the binary expansion of k is 0 and by a block Ω_T otherwise. In the former case, $\beta^j_{k,l} = \gamma^j_{k,l}$ iff v_j occurs negatively in C_l, in the latter case, $\beta^j_{k,l} = \gamma^j_{k,l}$ iff v_j occurs negatively in C_l. Thus, the principal type of V_j is indeed ϕ_j, and it can be derived in order 3 core-ML.

What remains is the construction of a term W that forces the types of $V_0, V_1, \ldots, V_n$ to be unified. W could be defined using the *Unify* operator introduced earlier, but to keep the order down, we use the following term:

$$W := \lambda f.\, K\,(f\,V_0)\,(K\,(f\,V_1)\ldots(K\,(f\,V_{n-1})\,(f\,V_n))\ldots).$$

Clearly, W forces the same type upon $V_0, V_1, \ldots, V_n$, and it can be constructed in polynomial time. As shown earlier, the types $\phi_0, \phi_1, \ldots, \phi_n$ unify if and only if Φ is not satisfiable, so W is ML-typable if and only if Φ is not satisfiable. If it is typable, then $V_0, V_1, \ldots, V_n$ have a common order 2 type derivable in order 3 core-ML, so the principal type of W can be derived in order 4 core-ML. Thus, the reduction is complete. □

7 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, but that type reconstruction is not significantly easier than the general case.

We would like to note that our characterization of FO-queries has some similarities with the extended polynomials characterization of [42]. Namely, the inputs and outputs have the same types. The only significant addition to TLC, with respect to [42], is equality.

Although we present a syntactically simple, functional characterization of FO- and PTIME-queries, our characterizations do depend on the input-output conventions. Different conventions might lead to other characterizations that might be of interest. Both the difference between o and τ and the treatment of duplicates are significant.

For example, if we allow duplicates in the input-output conventions and use Church numeral input-output instead of list-represented databases it is possible to build exponentiators of low functionality order, see [18].

For another example, in [26], we demonstrate how various non-FO-queries can be expressed with a slight variation of the framework. If, in addition to the typing $Eq : \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ prescribed

in Section 3, we also allow Eq to be typed as $Eq : \mathtt{o} \to \mathtt{o} \to \mathtt{o} \to \mathtt{o} \to \mathtt{o}$ (thereby introducing a weak form of polymorphism), we obtain a version $\mathrm{TLI}^{\simeq 0}$ of $\mathrm{TLI}_0^{=}$ that expresses relational algebra, parity, majority, and tree accessibility. $\mathrm{TLI}^{\simeq 0}$, together with a tupling construct, expresses exactly LOGSPACE-queries.

Questions: There are open issues regarding both variations of the input-output conventions (see above) and orders above 4. Beyond order 4 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 [27, 33, 35] to express various exponential time and space classes. Determining the exact expressive power of $\mathrm{TLI}_i^{=}$ and $\mathrm{MLI}_i^{=}$ for $i > 2$ is open; the case $i = 2$ was resolved in [24] and corresponds to PSPACE.

With respect to type reconstruction in core-ML, the order 3 case is open. For orders of 4 and above there is a gap between the best current upper bound (EXPTIME) and the NP-hardness lower bound.

A number of other interesting open problems are raised from the framework of this paper: (1) Study the use of optimal reduction strategies [37] for the evaluation of $\mathrm{TLI}_i^{=}$ queries. (2) Study languages that combine list iterators and set iterators ala [7, 8, 9, 30, 47]. (3) Study the expressive power and the complexity of type reconstruction for fixed-order fragments of other typed calculi, e.g., System F [19, 41].

Appendix: Relational Algebra in $\mathrm{TLI}_0^=$ and Copy in $\mathrm{TLI}_1^=$

The following encodings are based on the ones given in [25], adapted to the input-output format used in this paper. For a detailed explanation of their workings, consult [25]. Note that, if inputs are encodings without duplicates then so are outputs.

$$
\begin{array}{l}
\mathit{Setminus}_k : \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau := \\
\quad \lambda R : \mathsf{o}_k^\tau . \lambda S : \mathsf{o}_k^\tau . \\
\quad\quad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau . \lambda n : \tau . \\
\quad\quad\quad R\,(\lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o} . \lambda T : \tau . (\mathit{Member}_k\, x_1 \ldots x_k\, S)\, T\, (c\, x_1 \ldots x_k\, T))\, n
\end{array}
$$

$$
\begin{array}{l}
\mathit{Union}_k : \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau := \\
\quad \lambda R : \mathsf{o}_k^\tau . \lambda S : \mathsf{o}_k^\tau . \\
\quad\quad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau . \lambda n : \tau . \\
\quad\quad\quad R\,(\lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o} . \lambda T : \tau . (\mathit{Member}_k\, x_1 \ldots x_k\, S)\, T\, (c\, x_1 \ldots x_k\, T))\, (S\, c\, n)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Times}_{k,l} : \mathsf{o}_k^\tau \to \mathsf{o}_l^\tau \to \mathsf{o}_{k+l}^\tau := \\
\quad \lambda R : \mathsf{o}_k^\tau . \lambda S : \mathsf{o}_l^\tau . \\
\quad\quad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau . \lambda n : \tau . \\
\quad\quad\quad R\,(\lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o} . \lambda T : \tau . S\,(\lambda y_1 : \mathsf{o} \ldots \lambda y_l : \mathsf{o} . \lambda U : \tau . c\, x_1 \ldots x_k\, y_1 \ldots y_l\, U)\, T)\, n
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{Project}_{k;i_1,\ldots,i_l} : \mathsf{o}_k^\tau \to \mathsf{o}_l^\tau := \\
\quad \lambda R : \mathsf{o}_k^\tau . \\
\quad\quad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau . \lambda n : \tau . \\
\quad\quad\quad R\,(\lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o} . \lambda T : \tau . \\
\quad\quad\quad\quad R\,(\lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o} . \lambda S : \tau . \\
\quad\quad\quad\quad\quad \mathit{Equal}_l\, x_{i_1} \ldots x_{i_l}\, y_{i_1} \ldots y_{i_l}\, (\mathit{Equal}_k\, x_1 \ldots x_k\, y_1 \ldots y_k\, (c\, x_{i_1} \ldots x_{i_l}\, T)\, T)\, S)\, T)\, n
\end{array}
$$

Let $\chi := \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau \to \tau$, where o occurs k times in χ:

$$
\begin{array}{l}
\mathit{Copy}_i : \mathsf{o}_{k_i}^\chi \to \mathsf{o}_{k_i}^\tau := \\
\quad \lambda R : \mathsf{o}_{k_i}^\chi . \\
\quad\quad \lambda c : \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau . \lambda n : \tau . \\
\quad\quad\quad R\,(\lambda x_1 : \mathsf{o} \ldots \lambda x_{k_i} : \mathsf{o} . \lambda T : \chi . \lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o} . \lambda u : \tau . \lambda v : \tau . c\, x_1 \ldots x_{k_i}\, (T\, y_1 \ldots y_k\, u\, v)) \\
\quad\quad\quad (\lambda y_1 : \mathsf{o} \ldots \lambda y_k : \mathsf{o} . \lambda u : \tau . \lambda v : \tau . n) \\
\quad\quad\quad o_1 \ldots o_k\, n\, n \quad \text{(where } o_1, \ldots, o_k \text{ are arbitrary TLC}^= \text{ constants)}
\end{array}
$$

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, R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

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

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

[5] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics (revised edition).* North Holland, 1984.

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

[7] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the 3rd Workshop on Database Programming Languages (DBPL3)*, pp. 9–19. Morgan-Kaufmann, 1991.

[8] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *Proceedings of the 4th International Conference on Database Theory (ICDT)*, pp. 140–154. Lecture Notes in Computer Science 646, Springer Verlag, 1992.

[9] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *Proceedings of the 18th International Conference on Automata, Languages, and Programming (ICALP)*, pp. 60–75. Lecture Notes in Computer Science 510, Springer Verlag, 1991.

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

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

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

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

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

[15] 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.

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

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

[18] 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.

[19] J.-Y. Girard. *Interprétation Fonctionelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

[20] J.-Y. Girard, A. Scedrov, P. J. Scott. Bounded Linear Logic: a Modular Approach to Polynomial Time Computability *Theoretical Comp. Sci.*, **97** (1992), pp. 1–66.

[21] C. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

[22] Y. Gurevich. Algebras of Feasible Functions. In *Proceedings of the 24th IEEE Conference on the Foundations of Computer Science (FOCS)* (1983), pp. 210–214.

[23] R. Harper, R. Milner, M. Tofte. *The Definition of Standard ML.* MIT Press, 1990.

[24] G. Hillebrand. *Finite Model Theory in the Simply Typed Lambda Calculus.* Ph.D. thesis, Brown University, 1994.

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

[26] G. Hillebrand and P. Kanellakis. Functional Database Query Languages as Typed Lambda Calculi of Fixed Order. In *Proceedings of the 13th ACM Symposium on the Principles of Database Systems (PODS)*, pp. 222–231, 1994.

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

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

[29] N. Immerman. Expressibility and Parallel Complexity *SIAM J. Comp.*, **18** (1986), pp. 625–638.

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

[31] 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.

[32] A. Kfoury, J. Tiuryn, and P. Urzyczyn. An Analysis of ML Typability. In *Proceedings of the 17th Colloquium on Trees, Algebra and Programming*, pp. 206–220. Lecture Notes in Computer Science 431, Springer Verlag, 1990.

[33] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The Hierarchy of Finitely Typed Functional Programs. In *Proceedings of the 2nd IEEE Conference on Logic in Computer Science (LICS)* (1987), pp. 225–235.

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

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

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

[37] 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.

[38] H. Mairson. A Simple Proof of a Theorem of Statman. *Theoretical Comput. Sci.*, **103** (1992), pp. 387–394.

[39] A. R. Meyer. The Inherent Computational Complexity of Theories of Ordered Sets. In *Proceedings of the International Congress of Mathematicians*, pp. 477–482, 1975.

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

[41] J. Reynolds. Towards a Theory of Type Structure. In *Proceedings of the Paris Colloquium on Programming*, pp. 408–425. Lecture Notes in Computer Science 19, Springer Verlag, 1974.

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

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

[44] R. Statman. Completeness, Invariance, and λ-Definability. *J. Symbolic Logic*, **47** (1982), pp. 17–26.

[45] R. Statman. Equality between Functionals, Revisited. In *Harvey Friedman's Research on the Foundations of Mathematics*, pp. 331–338. North-Holland, 1985.

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

[47] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of the 12th ACM Symposium on the Principles of Database Systems (PODS)* (1993), pp. 26–36.