Database Query Languages Embedded in the Typed Lambda Calculus*

Gerd G. Hillebrand† Paris C. Kanellakis‡ Harry G. Mairson§

Abstract

We investigate the expressive power of the typed λ-calculus when expressing computations over finite structures, i.e., databases. We show that the simply typed λ-calculus can express various database query languages such as the relational algebra, fixpoint logic, and the complex object algebra. In our embeddings, inputs and outputs are λ-terms encoding databases, and a program expressing a query is a λ-term which types when applied to an input and reduces to an output. Our embeddings have the additional property that PTIME computable queries are expressible by programs that, when applied to an input, reduce to an output in a PTIME sequence of reduction steps. Under our database input-output conventions, all elementary queries are expressible in the typed λ-calculus and the PTIME queries are expressible in the order-5 (order-4) fragment of the typed λ-calculus (with equality).

1 Introduction

TLC Motivation and Background: The *simply typed λ-calculus* of Church [12] (*typed λ-calculus* or TLC for short) with its syntax and operational semantics is an essential part of any functional programming language. For example, TLC is a core subset of all state-of-the-art functional languages such as ML, Haskell, Miranda, etc. TLC together with `let`-polymorphism [22, 38] is often informally referred to as core-ML. In this paper, we investigate the expressive power of TLC from the point of view of expressing computations over finite structures. In other words, we study the ability of TLC to express database queries.

Our interest in database computations is in marked contrast to the classical approach to TLC expressibility, which considers computations over Church numerals (see, e.g., [4, 16, 42]). There are several results characterizing the expressive power of TLC over Church numerals, but the picture is somewhat complex. If inputs and outputs are Church numerals typed as Int (where $\mathsf{Int} \equiv (\tau \to \tau) \to \tau \to \tau$ for some fixed τ), Schwichtenberg [42] and Statman showed that the expressible multi-argument functions of type $(\mathsf{Int}, \ldots, \mathsf{Int}) \to \mathsf{Int}$ (or equivalently, $\mathsf{Int} \to \cdots \to \mathsf{Int} \to \mathsf{Int}$) are exactly the *extended polynomials*, i.e., the functions generated by 0 and 1 using the operations addition, multiplication and conditional. If inputs and outputs are Church numerals given more complex types than Int, exponentiation and predecessor can also be expressed. However, Statman (as quoted in [16]) showed that equality, ordering, and subtraction are not expressible in TLC for any typing of Church numerals.

*A preliminary version of the work presented here appeared in [25].

†Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

‡Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

§Dept. of Computer Science, Brandeis University, Waltham, MA 02254. Research supported by NSF Grant CCR-9216185, ONR Grant N00014-93-1-1015, and the Tyson Foundation.

These classical expressibility results have cast a negative and slightly confusing light on the possible encodings of computations in TLC. Simple types have been criticized for limiting flexibility in programs, but they have also been criticized for limiting expressibility. These criticisms have provided some motivation for examining more powerful typed calculi, such as the Girard-Reynolds second-order λ-calculus [17, 41] (adding polymorphism via type quantification) or Milner's ML [22, 38] (adding monomorphic fixpoints and `let`-polymorphism). We believe that the criticism of TLC inflexibility is justified, although hard to quantify. The criticism of TLC expressibility is unjustified, and a theme of this paper is to quantify how rich a framework TLC is for expressing computations, provided that the right setting is chosen.

In fact, it is well known that provably hard decision problems can be embedded into TLC. This follows from a theorem of Statman that deciding equivalence of normal forms of two well-typed λ-terms is not elementary recursive [43]. The proof in [43] uses a result of Meyer concerning the complexity of decision problems in higher-order type theory [37]. A simple proof of both these results appears in [35]. However, there are a number of difficulties when one tries to turn these proofs into frameworks for computations. They do not separate the fixed *program* (representing a function) from the variable *data* (representing the input). They use computational overkill for lower complexity classes. Specifically, the *Powerset* construction, crucial to all of the proofs, adds exponential factors to the computation. For example, a simulation of quadratic time is forced to take at least an exponential number of reduction steps in these constructions.

One way of avoiding the anomalies associated with representations over Church numerals was recently demonstrated by Leivant and Marion [33] for an "impure" version of TLC. By augmenting the simply typed λ-calculus with a pairing operator and a "bottom tier" consisting of the free algebra of words over $\{0, 1\}$ with associated constructor, destructor, and discriminator functions, they obtain a simple characterization of the computational complexity class PTIME.

In this paper we re-examine the question of representing functions in the "pure" TLC, as opposed to "impure" versions. However, we use encodings of *finite models* or *finite first-order relational structures* (*databases* for short) instead of Church numerals. Thus, we are changing the problem from encoding numerical functions to encoding generic set functions, i.e., *database queries*. For our input and output databases we use standard techniques of encoding lists and tuples in the typed λ-calculus. Queries are then encoded as well-typed λ-terms that apply to encodings of input relations and reduce to an encoding of the output relation. For notational convenience, we use TLC$^=$, the typed λ-calculus with atomic constants and an equality on them, and the associated δ-reduction of [4, 12]. This is not essential for our analysis. In Section 2.4, we show how to encode atomic constants and equality in TLC.

Our change of data representation, i.e., the framework of finite model theory instead of arithmetic on Church numerals, has some interesting consequences, because it takes us into the realm of *database query languages*.

DB Motivation and Background: Database query languages have been motivated by Codd's work on *relational databases* [14] and have been studied in the context of *finite model theory*, e.g., [9, 10, 11, 15, 27, 46, 48]. Database queries, i.e., the functions from finite models to finite models expressed in various languages, have been classified based on their complexity. The most commonly used measure is the one of *data complexity* of [10, 48], where the program expressing a query is fixed and the input data is variable. For example, the PTIME queries are those with data complexity that is polynomial in the size of the input.

Relational algebra [14], *Datalog* and *Datalog with Negation* [30] (written as *Datalog*$^\neg$), and various *fixpoint logics* [3, 10, 11, 20, 32] express practically interesting sets of database queries, all subsets of the PTIME queries. In addition, as shown in [27, 48], every PTIME query can be expressed using fixpoint queries on ordered structures; and, as shown in [3], it suffices to use Datalog$^\neg$ syntax under a variety of semantics (e.g., inflationary) to express the various fixpoint logics.

Complex object databases have been proposed as a significant extension of relational databases, with many practical applications; see [2] for a recent survey. Well-known languages in this area are the *complex object algebras* with or without *Powerset* of [1]. For the analysis of expressibility of the complex object algebra with *Powerset* we refer to [26, 31] and without *Powerset* to [39]. Note that, although *Powerset* in [1] is powerful (as are the second order queries in [10, 15, 46]), it is an impractical primitive, and much attention has been given to algebras without *Powerset* for PTIME queries.

An elegant way of manipulating complex object databases, related to our paper, is based on functional programming. There has been some practical work in database query languages in this area, e.g., the early FQL language of [8] and the more recent work on structural recursion as a query language [6, 7, 28]. One important difference of the framework developed here from [6, 7, 28] is that we use the "pure" TLC without any added recursion operators (the equality predicate and atomic constants used in our presentation are not essential for our results).

Contributions: The topic of this paper is how to embed database query languages in the typed λ-calculus with (and without) atomic constants and an equality predicate on these constants. We consider three requirements:

1. inputs and outputs are λ-terms encoding finite sets of tuples for relational databases, or more arbitrary combinations of finite sets and tuples for complex object databases,
2. programs are λ-terms which type when applied to an input, reduce to an output, and express input-output functions that are database queries of interest,
3. there is a reduction strategy for the application of the program to the input, such that the output normal form is produced efficiently by the reductions (i.e., in time polynomial in the size of the input) when the functions expressed are PTIME queries.

Requirements (1–2) above give us *embeddings*, in the sense of expressing queries of interest. The additional requirement (3) is important if one wishes to consider the typed λ-calculus as a functional database query language operating by reduction. We call embeddings that satisfy (1–3) *PTIME-embeddings*.

It is implicit in the literature [35, 37, 43] that, under our input-output conventions but without considering an efficient reduction strategy, all elementary functions are expressible (where this class of functions includes PTIME, NP, PSPACE, EXPTIME, k-EXPTIME, etc. [40]). For all practical purposes, ELEMENTARY is a powerful complexity class with a somewhat misleading name; to quote from [40]: "the optimism in this term may seem a little overstated; the term was introduced in the context of undecidability." Thus, our finite model input-output conventions illustrate the nontrivial power of TLC.

Our new contribution here is that we provide PTIME-embeddings of various practical query languages. The main result of this paper is: *It is possible to PTIME-embed in the typed λ-calculus with equality the following database query languages: the relational algebra, inflationary Datalog with negation, and the complex object algebra.* For all these languages we can adapt the framework to eliminate atomic constants and equality (see Section 2.4).

In all the PTIME-embeddings of this paper, the reduction strategies are described as part of the proofs and are simple (in fact, we use "eager" reduction in all but a few cases). They are easily implementable in PTIME on any Turing Machine implementation of TLC.

We also study syntactically restricted fragments of the full TLC defined using the standard notion of functionality order (cf. Section 2.1). Using the characterization of PTIME by [27, 48] and a slightly modified encoding of databases, we prove: *It is possible to embed all PTIME queries in order 4 TLC$^=$*. Here we must keep equality as part of the setting, i.e., its removal would raise the order. *Without equality, it is possible to embed all PTIME queries in order 5 TLC.* However, these embeddings are not PTIME-embeddings in the sense above, because they do not come with PTIME reduction strategies. Turning them into PTIME-embeddings requires a change in the computational engine, i.e., the TLC reduction mechanism (this is carried out in [23, 24]).

The *deus ex machina* of this paper is list iteration—primitive recursion on lists. While certainly less powerful than unbounded recursion, it does what we need. The list iteration technology developed in our proofs, e.g., duplicate elimination in relational algebra or implementing iteration in Datalog is interesting. Also, the "type-laundering" constructions are somewhat involved. `let`-polymorphism would have greatly simplified them, but would have taken us outside TLC.

From a programmer's point of view, we illustrate how simply typed LISP [36] can accomplish a great deal with *lambda*, even without using recursion on names or fixpoint combinators!

From a database query language perspective, our PTIME-embeddings indicate that the typed λ-calculus, with its syntax, semantics and reduction strategies, can be viewed as a unifying functional framework that is between the declarative calculi and the procedural algebras. After all it is called a calculus, but reductions are procedural.

Organization of the paper: In Section 2, we set out our framework. Section 2.1 presents the necessary background on the "pure" TLC and on an "impure" variant with equality, TLC$^=$. Section 2.2 outlines the basics of list iteration. Section 2.3 describes our input-output conventions with atomic constants and equality in detail, and Section 2.4 discusses how to encode atomic constants and equality in the "pure" TLC.

In Section 3, we PTIME-embed relational algebra (and by [14], relational calculus) into TLC$^=$ by encoding the relational algebra operators. The encodings are mostly straightforward, but some care is needed to eliminate duplicate tuples from the output.

In Section 4, we PTIME-embed Datalog$^\neg$ into TLC$^=$ by encoding fixpoint queries. We first iterate relational algebra expressions without satisfying all typing requirements in Section 4.1. The typing requirements are taken care of in Section 4.2, using a technique one can describe as "type-laundering." To illustrate the embedding, Section 4.3 contains the representation of the transitive closure query as a detailed example.

In Section 5, we PTIME-embed the complex object algebra (and by [1], the complex object calculus) into TLC$^=$. We first present a concise definition of this algebra from [2] (Section 5.1). There are two tasks. The first is to PTIME-embed the algebra without *Powerset*; this is accomplished in Section 5.2. The second is to embed the *Powerset* operator; this is accomplished in Section 5.3 using additional "type-laundering" technology.

In Section 6, we modify our earlier embeddings to minimize the functionality order. This leads to an embedding of QPTIME (the class of all PTIME queries) in order 4 TLC$^=$ and order 5 TLC.

We conclude with some open problems in Section 7.

2 The Typed λ-Calculus with Equality (TLC$^=$)

2.1 TLC and TLC$^=$ Syntax and Semantics

TLC: The syntax of TLC *types* is given by the grammar $\mathcal{T} \equiv \zeta \mid (\mathcal{T} \to \mathcal{T})$, where ζ ranges over a set of *type variables* $\{\rho, \sigma, \tau, \ldots\}$. For example, ρ is a type, as are $(\rho \to \sigma)$ and $(\rho \to (\rho \to \rho))$. In the following, $\alpha, \beta, \gamma, \ldots$ denote types. We omit outermost parentheses and write $\alpha \to \beta \to \gamma$ for $\alpha \to (\beta \to \gamma)$.

The syntax of TLC *terms* or *expressions* is given by the grammar $\mathcal{E} \equiv \xi \mid (\mathcal{E}\mathcal{E}) \mid \lambda\xi.\,\mathcal{E}$, where ξ ranges over a set of *expression variables* $\{x, y, z, \ldots\}$ and where expressions are *well-typed* as outlined below. In the following, $E, F, G \ldots$ denote expressions. We omit outermost parentheses and write EFG for $(EF)G$.

Typability of expressions is defined by the following inference rules, where Γ is a function from expression variables to types, and $\Gamma + \{x{:}\,\alpha\}$ is the function Γ' updating Γ with $\Gamma'(x) = \alpha$:

(Var) $$\frac{}{\Gamma + \{x{:}\,\alpha\} \vdash x{:}\,\alpha}$$

(Abs) $$\frac{\Gamma + \{x{:}\,\alpha\} \vdash E{:}\,\beta}{\Gamma \vdash \lambda x.\,E{:}\,\alpha \to \beta}$$

(App) $$\frac{\Gamma \vdash E{:}\,\alpha \to \beta \quad \Gamma \vdash F{:}\,\alpha}{\Gamma \vdash (EF){:}\,\beta}$$

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

The operational semantics of TLC are defined using *reduction*. For well-typed λ-terms E, E', we write $E \rhd_\alpha E'$ (α-reduction) when E' can be derived from E by renaming of a λ-bound variable, for example $\lambda x.\,\lambda y.\,y \rhd_\alpha \lambda x.\,\lambda z.\,z$. We write $E \rhd_\beta E'$ (β-reduction) when E' can be derived from E by replacing a subterm in E of the form $(\lambda x.\,F)\,G$ (called a *redex*) by $F\,[x{:=}\,G]$ (F with G substituted for all free occurrences of x in F). In the following, we identify α-convertible terms and assume that names of bound variables are chosen to be distinct from each other and from all free variables. See [4] for details of this "variable convention", standard definitions of substitution and α- and β-reduction for both the typed and untyped λ-calculus, and other reduction notions such as η-reduction.

Let $\rhd$ be the reflexive, transitive closure of $\rhd_\beta$. Note that, reduction preserves types.

Curry vs Church Notation: In the above definition, we have adopted the "Curry View" of TLC, where types can be *reconstructed* for unadorned terms using the inference rules. We could have chosen the "Church View," where types and terms are defined together and λ-bound variables are annotated with their type (i.e., we would have $\lambda x{:}\,\alpha.\,E$ instead of $\lambda x.\,E$). In our encodings below we usually provide "Church style" type annotations to prove that the terms are indeed well-typed.

TLC$^=$: We obtain TLC$^=$ by enriching TLC with: (1) a type constant o, (2) a countably infinite set $\{o_1, o_2, \ldots\}$ of expression constants of type o, and (3) an expression constant Eq of type $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ (where τ is some fixed type variable). The type inference rules for TLC$^=$ are those of TLC augmented with axioms $o_i{:}\,\mathsf{o}$ ($i = 1, 2, \ldots$) and $Eq{:}\,\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$. 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, we add to $\rhd$ the reduction rule (known as a δ-reduction rule):

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

and then close $\triangleright$ transitively again. Note that this augmented notion of reduction also preserves types. The motivation behind the *Eq* reduction rules is the desire to express statements of the form "if $x = y$ then E else F"—with the above rules, this can be written simply as $(Eq\, x\, y\, E\, F)$.

A λ-term from which no reduction is possible is in *normal form*. TLC and TLC$^=$ enjoy the following properties, see [4, 21]:

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

2. *Strong normalization property:* For each E, there exists an integer n such that if $E \triangleright E'$, then the derivation involves no more than n individual reductions.

3. *Principal type property:* A well-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 well-typed λ-term and the principal type of this term can be reconstructed. Also, given $\Gamma \vdash E : \alpha$, 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). TLC and TLC$^=$ type reconstruction is *linear-time* in the size of the program analyzed.

5. *Completeness of Equational Reasoning:* $\beta\eta$-equality completely characterizes validity in TLC models. See [21, 44, 45] for details and for many semantic properties of TLC.

Functionality Order: The *order* of a type, which measures the higher-order functionality of a λ-term of that type, is defined as $\mathrm{order}(\alpha) = 0$, if α is a type variable or type constant, and $\mathrm{order}(\alpha) = \max(1 + \mathrm{order}(\beta), \mathrm{order}(\gamma))$, if α is an arrow type $\beta \to \gamma$. We also refer to the *order of a well-typed λ-term* as the order of its principal type. 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 List Iteration: Some Examples

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

$$L \equiv \lambda c.\, \lambda n.\, c\, E_1\, (c\, E_2 \ldots (c\, E_k\, n) \ldots)$$

is a well-typed λ-term with principal type $(\alpha \to \rho \to \rho) \to \rho \to \rho$, where ρ is a type variable. We abbreviate this list construction as $[E_1, E_2, \ldots, E_k]$; the variables c and n abstract over the list constructors *Cons* and *Nil*. In functional programming terminology, $[E_1, E_2, \ldots, E_k]$ is a partially evaluated *foldr* operator with its list argument fixed.[1]

To understand how list iteration works, think of L as a "do"-loop defined in terms of a "loop body" c and a "seed value" n. The loop body is invoked once for every element in L, starting from the last and proceeding backwards to the first. (By Church-Rosser and strong normalization, all

[1]If we had written L as $\lambda c.\, \lambda n.\, (c\, (\ldots (c\, (c\, n\, E_1)\, E_2) \ldots)\, E_k)$, we would have obtained a partially evaluated *foldl* operator. Both representations are equivalent in terms of expressive power, since we can go from one to the other by reversing the order of $E_1, E_2, \ldots, E_k$ and swapping the arguments of c.

evaluation orders lead to the same results.) 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 L. Once all elements have been processed, the final value of the accumulator is returned.

As an example, consider the problem of determining the parity of a list of Boolean values. A standard encoding of Boolean logic uses $\mathit{True} \equiv \lambda x{:}\,\tau.\, \lambda y{:}\,\tau.\, x$ and $\mathit{False} \equiv \lambda x{:}\,\tau.\, \lambda y{:}\,\tau.\, y$, both of type $\mathsf{Bool} \equiv \tau \to \tau \to \tau$. The exclusive-or function can be written as

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

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:

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

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.

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.

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

(1) A list L of Booleans has principal type $\beta \equiv (\mathsf{Bool} \to \rho \to \rho) \to \rho \to \rho$, where ρ is a type variable. To type $\mathit{Parity}\, L$, we must type L with $\mathsf{Bool} \equiv \tau \to \tau \to \tau$ substituted for ρ, which we write as $\beta\,[\rho{:=}\ \mathsf{Bool}]$. Similarly, to type $\mathit{Length}\, L$, we need to type L as $\beta\,[\rho{:=}\ \mathsf{Int}]$. Thinking of the principal type β of L as a "clean" type, in both instances we see that the type of L must be "contaminated" or "raised" to provide the primitive recursive iterator. If we want to use L in two different contexts requiring different substitution instances of its principal type, this poses a problem. The problem can be solved by defining a "type-laundering" operator that transforms a "contaminated" list, e.g., $L{:}\,\beta\,[\rho{:=}\ \mathsf{Int}]$, into a "clean" list of type just β.

(2) If we do not want to "contaminate" the type at all, this poses yet another difficulty. This problem can sometimes be handled by "type-clean" encodings. Revisiting the length example we can use $\mathit{Length} \equiv \lambda l{:}\,(\alpha \to \rho \to \rho) \to \rho \to \rho.\, \lambda s{:}\,\rho \to \rho.\, \lambda z{:}\,\rho.\, l\,(\lambda x{:}\,\alpha.\, s)\, z$, where l is used to iterate over objects of lower type.

Both "type-laundering" and "type-clean" encoding techniques are exploited repeatedly in the following sections.

2.3 Databases as λ-Terms

We compute on finite models. Inputs and outputs of a computation are *finite models* or *databases*, which are encoded according to Definition 2.1. To motivate the encoding conventions postulated there, let us illustrate step by step how to build relational data structures in TLC$^=$.

Let $\mathcal{D} = \{o_1, o_2, \ldots\}$ be the set of atomic constants, of type o, of the TLC$^=$ calculus. We assume that the same set of constants also serves as the universe over which all finite models are defined. Let τ be the fixed type variable used in the typing $Eq : \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$.

Boolean values are represented by

$$True : \tau \to \tau \to \tau \equiv \lambda u : \tau.\, \lambda v : \tau.\, u, \quad False : \tau \to \tau \to \tau \equiv \lambda u : \tau.\, \lambda v : \tau.\, v.$$

We abbreviate the type $\tau \to \tau \to \tau$ as Bool. Note that under this convention the type of Eq becomes $\mathsf{o} \to \mathsf{o} \to \mathsf{Bool}$ and the reduction rules for Eq assume the more transparent form $(Eq\, o_i\, o_j) \rhd True$ if $i = j$, $(Eq\, o_i\, o_j) \rhd False$ if $i \neq j$. Also, if E is of type Bool and F and G are of type τ, we can write $(E\, F\, G)$ to simulate the conditional "if E then F else G."

Tuples are represented as follows: if $E_1, \ldots, E_k$ are λ-terms of type $\alpha_1, \ldots, \alpha_k$, then

$$\langle E_1, \ldots, E_k \rangle : (\alpha_1 \to \cdots \to \alpha_k \to \tau) \to \tau \equiv \lambda f : \alpha_1 \to \cdots \to \alpha_k \to \tau.\, f\, E_1 \ldots E_k$$

We abbreviate the type $(\alpha_1 \to \cdots \to \alpha_k \to \tau) \to \tau$ as $\alpha_1 \times \cdots \times \alpha_k$ or, if $\alpha_i \equiv \alpha$ for $1 \leq i \leq k$, as α^k. In particular, o^k is the abbreviation for the type of a k-tuple of atomic constants. Note that α^1 is different from α: it denotes the type $(\alpha \to \tau) \to \tau$. The representation of tuples is chosen to make it easy to extract their components: the i^{th} component of a k-tuple T is produced by the expression $(T\, (\lambda x_1 \ldots \lambda x_k.\, x_i))$, as the reader may verify.

Lists are represented as described above: if $E_1, \ldots, E_k$ are λ-terms of type α, then

$$[E_1, \ldots, E_k] : (\alpha \to \tau \to \tau) \to \tau \to \tau \equiv \lambda c : \alpha \to \tau \to \tau.\, \lambda n : \tau.\, c\, E_1\, (c\, E_2 \ldots (c\, E_k\, n) \ldots)$$

We abbreviate the type $(\alpha \to \tau \to \tau) \to \tau \to \tau$ as $\{\alpha\}$. Substitution instances of this type, where a type β has been substituted for τ, are written as $\{\alpha\}\,[\tau := \beta]$. Note the difference between lists and tuples: a tuple represents a collection of a fixed number of terms of varying type, whereas a list represents a collection of a variable number of terms of a fixed type.

Relations are represented as duplicate-free lists of tuples of constants. An example encoding of a relation is given in Definition 2.1 below. We usually type a k-ary relation as $\{\mathsf{o}^k\}$, but sometimes we use a "contaminated" type $\{\mathsf{o}^k\}\,[\tau := \alpha]$, where α is some type expression.

Queries are represented by TLC$^=$ terms with the property that, when applied to encodings of (the proper number of) input relations (of the proper arities), the combined term: (1) types, (2) reduces to an encoding of the desired output relation, and (3) produces the same output relation independent of the order in which tuples are listed in the input encodings. Our encodings of queries make sure that their output is duplicate-free, provided that the input is duplicate-free.

Let us state our encoding conventions precisely:

Definition 2.1 *Let $R = \{(o_{1,1}, o_{1,2}, \ldots, o_{1,k}), (o_{2,1}, o_{2,2}, \ldots, o_{2,k}), \ldots, (o_{m,1}, o_{m,2}, \ldots, o_{m,k})\} \subseteq \mathcal{D}^k$ be a k-ary relation over $\mathcal{D}$. An* encoding *of R, denoted by $\overline{R}$, is a λ-term:*

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

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.

Let $(k_1,\ldots,k_l,k)$ be a set of arities. A query term *of arity $(k_1,\ldots,k_l,k)$ is a TLC$^=$ term Q with the following properties.*

1. Typability: *For all encodings $\overline{R_1},\ldots,\overline{R_l}$ of relations of arities $k_1,\ldots,k_l$ over $\mathcal{D}$, the term $(Q\,\overline{R_1}\ldots\overline{R_l})$ is well-typed in TLC$^=$.*

2. Well-formed output: *For all encodings $\overline{R_1},\ldots,\overline{R_l}$ of relations of arities $k_1,\ldots,k_l$ over $\mathcal{D}$, the term $(Q\,\overline{R_1}\ldots\overline{R_l})$ reduces to a normal form $\overline{R}$ which is the encoding of a k-ary relation R.*

3. Encoding independence: *If $\overline{R_1},\ldots,\overline{R_l}$ and $\overline{R_1}',\ldots,\overline{R_l}'$ are encodings of the same input relations $R_1,\ldots,R_l$, then the normal forms of $(Q\,\overline{R_1}\ldots\overline{R_l})$ and $(Q\,\overline{R_1}'\ldots\overline{R_l}')$ are encodings of the same output relation R. We call R the* output *of Q on inputs $R_1,\ldots,R_l$, and we call the mapping $(R_1,\ldots,R_l)\mapsto R$ the database query* expressed *by Q.*

This definition will be used in our embeddings of relational algebra and fixpoint queries in the next two sections. We will generalize it for the complex object algebra so as to encode complex object databases (in Section 5) and modify it for minimizing the functionality order (in Section 6).

Let us comment on a few aspects of our definition of query terms.

Syntactic vs. semantic conditions: The above definition of query terms is semantic in the sense that its three conditions should hold for all encodings of relations over $\mathcal{D}$ of arities $k_1,\ldots,k_l$. This raises the question: Given arities $(k_1,\ldots,k_l,k)$ and a TLC$^=$ term Q, is Q a query term? In general, this is undecidable. However, for the embeddings constructed in the next sections, the conditions are easily verified and this is part of our proofs. One might also ask whether conditions (1)–(3) can be made syntactic by translating them into constraints on the type of Q. For condition (1), this is true, because the typing of a term $(Q\,\overline{R_1}\ldots\overline{R_l})$ is essentially independent of the size of $R_1,\ldots,R_l$. For conditions (2) and (3), this fails, because encodings of relations cannot be characterized by their types (there are TLC$^=$ terms of type $\{\mathsf{o}^k\}$ that are not encodings of relations, and different encodings of the same relation are indistinguishable at the type level).

Genericity: Without condition (3), a query term would only define a mapping from encodings of relations to encodings of relations, not necessarily from relations to relations. It is possible to write λ-terms that satisfy conditions (1) and (2), but not (3) (cf. the *First* operator used in the implementation of projection in Section 3). This seems inherent in our framework, since the λ-calculus does not provide unordered sets as primitive objects. Note that, condition (3) and the fact that atomic constants are not distinguishable by the *Eq*-reduction rules, enforce genericity in the sense of [10]. To see this, consider a query term Q, input relations $R_1,\ldots,R_l$ for Q, and a bijection ϕ of $\mathcal{D}$ that acts as the identity on the constants appearing in Q. Let R be the relation encoded by the normal form of $(Q\,\overline{R_1}\ldots\overline{R_l})$, $\phi(R_i)$ be the image of R_i under the canonical extension of ϕ to structures over $\mathcal{D}$, and $\phi(\overline{R_i})$ be the TLC$^=$ term that arises from $\overline{R_i}$ by replacing every constant with its image under ϕ. Observe that $\phi(\overline{R_i})$ is an encoding of $\phi(R_i)$, so by condition (3), the output of Q on inputs $\phi(R_1),\ldots,\phi(R_l)$ is encoded by the normal form of $(Q\,\phi(\overline{R_1})\ldots\phi(\overline{R_l}))$. Consider now a reduction sequence leading from $(Q\,\overline{R_1}\ldots\overline{R_l})$ to its normal form $\overline{R}$. If we replace every constant occurring in this reduction sequence by its image under ϕ, we obtain a valid reduction sequence leading from $\phi(Q\,\overline{R_1}\ldots\overline{R_l})\equiv(Q\,\phi(\overline{R_1})\ldots\phi(\overline{R_l}))$ to $\phi(\overline{R})$. (This is because different constants have different images under ϕ, so the outcome of any *Eq*-reductions is unaffected by ϕ.) It follows that the output of Q on inputs $\phi(R_1),\ldots,\phi(R_l)$ is given by $\phi(R)$, so Q represents a generic query.

Typing inputs and outputs: The definition of a query term requires that for legal inputs the program type checks and produces a legal output in a generic way. In imposing these conditions, we are as flexible as possible with the typings of legal inputs and outputs. In particular, we allow the types of inputs and outputs to differ. For example, the type assigned to an input $\overline{R_i}$ in a particular query $(Q\,\overline{R_1}\ldots\overline{R_l})$ may be of the form $\{\mathsf{o}^{k_i}\}\,[\tau := \alpha_i]$, where α_i is some type expression, whereas the type assigned to an output $\overline{R}$ may be of the form $\{\mathsf{o}^k\}$. This is different from the setting of [16, 42], but necessary to express queries beyond relational algebra. We type k-ary relations as $\{\mathsf{o}^k\}$ for inputs and outputs in Section 3 and for outputs in Section 4. We use the less restrictive typing $\{\mathsf{o}^k\}\,[\tau := \alpha]$, where α is some type expression, for inputs in Section 4. The typing of inputs and outputs for Section 5 is explained in detail in the description of the *Powerset* operator there.

When a query term Q expresses a database query q, we say that q can be *embedded* into TLC$^=$. This is purely a measure of expressive power. In practice, we also want efficient embeddings, in the sense that PTIME queries should be expressed by λ-terms that can be evaluated in polynomial time. However, computation in TLC$^=$ is a nondeterministic reduction process that does not immediately correspond to a machine model, so it is not obvious what is meant by "can be evaluated in polynomial time."

One desirable condition is that there exists a sequence of reduction steps that produces the normal form in a polynomial number of steps using intermediate terms of at most polynomial size. In this case, a nondeterministic Turing machine that "guesses" the right redexes to contract can compute the normal form in polynomial time, since each reduction step requires at most a polynomial number of moves. We adopt the following deterministic version of this (weaker) existential condition.

Definition 2.2 *A* reduction strategy ψ *is a mapping from non-normal-form λ-terms to occurrences of redexes in these terms. A reduction sequence is carried out* according to ψ *if for each reduction step $E \triangleright E'$ in the sequence, the redex contracted is the one given by $\psi(E)$. Given a λ-term Q and a family of λ-terms $\mathcal{I} = \{I_1, I_2, \ldots\}$ such that $(Q\,I_j)$ is well-typed for all j, ψ is a* PTIME reduction strategy for Q and $\mathcal{I}$ *if:*

1. *The normal form of $(Q\,I_j)$ can be computed in a number of reduction steps polynomial in the size of I_j if the reductions are carried out according to ψ.*

2. *All intermediate terms occurring during the reduction of $(Q\,I_j)$ in (1) above are of size at most polynomial in the size of I_j.*

3. *ψ is computable, and the computation of $\psi(E)$ for all intermediate terms E occurring during the reduction of $(Q\,I_j)$ can be carried out in time polynomial in the size of I_j.*

Definition 2.3 *Let q be a PTIME database query mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. A* PTIME-embedding *of q into TLC$^=$ is a pair (Q, ψ), where Q is a TLC$^=$ query term of arity $(k_1, \ldots, k_l, k)$ expressing q (see Definition 2.1) and ψ is a PTIME reduction strategy for Q and the family of encodings of relations of arities $k_1, \ldots, k_l$ (see Definition 2.2).*

PTIME-embeddings are interesting because they capture efficient functional (i.e., reduction) computations. For such embeddings, the time needed to evaluate a query by executing the reduction sequence on a sequential machine (such as a Turing Machine implementing TLC$^=$ reduction) is polynomial in the size of the inputs. The PTIME-embeddings we present in the following sections in fact all use very simple reduction strategies (except for a few cases, they use "eager" or "call-by-value" reduction, where a redex $(\lambda x.\, M)\, N$ is contracted only after N has been fully reduced), and their running time on a sequential machine typically matches that of a naive implementation of the corresponding queries in a procedural language.

2.4 Encoding Constants and Equality

As mentioned in the introduction, the presence of atomic constants and an equality predicate is not essential for the expressive power of TLC. We present here a simple way of encoding these "impure" features using "pure" TLC terms. For another way of coding up equality on finite domains see [35].

Let $\mathcal{A} = \{o_1, o_2, \ldots, o_N\}$ be the set of constants occurring in a particular input. We call this finite set the *active domain*. Given $\mathcal{A}$, we code the constant o_i as the projection function

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

typable as $\omega \equiv \tau \to \cdots \to \tau$, where the number of τ's is $N+1$ and depends on the size of the active domain. The equality predicate is coded as the term

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

typable as $\omega \to \omega \to \mathsf{Bool}$ which, when applied to two projection functions π_i^N and π_j^N, reduces to $\lambda u.\, \lambda v.\, u$ if $i = j$ and to $\lambda u.\, \lambda v.\, v$ otherwise.

The input is then encoded in the usual way, except that database constants are replaced by their corresponding projection functions. Since the encoding of Eq now depends on the size of the active domain, Eq has to be part of the input. So Eq has to be λ-bound at the outermost level of the query term.

With these modifications, query terms described in the following sections express the same queries and can be evaluated with similar PTIME reduction strategies. They are typable in TLC as they are in TLC$^=$ with ω instead of o. Thus, the conditions of Definitions 2.1 and 2.3 are still satisfied (at the price of having the typings depend in a uniform fashion on the input size).

3 Coding Relational Algebra

We begin by demonstrating how the relational algebra operators of [14] can be represented in the simply typed λ-calculus with equality. This involves coding the following operators, where t, t' denote tuples and R, S relations, that is, sets of tuples (see [47] for more details):

- *Intersection*, defined by $\mathit{Intersection}\,(R,S) = \{t \mid t \in R \wedge t \in S\}$, where $\mathrm{arity}\,(R) = \mathrm{arity}\,(S)$;
- *Setminus*, defined by $\mathit{Setminus}\,(R,S) = \{t \mid t \in R \wedge t \notin S\}$, where $\mathrm{arity}\,(R) = \mathrm{arity}\,(S)$;
- *Union*, defined by $\mathit{Union}\,(R,S) = \{t \mid t \in R \vee t \in S\}$, where $\mathrm{arity}\,(R) = \mathrm{arity}\,(S)$;
- *Times*, defined by $\mathit{Times}\,(R,S) = \{t\,t' \mid t \in R \wedge t' \in S\}$, where $t\,t'$ denotes the concatenation of t and t';
- Select_ϕ, defined by $\mathit{Select}_\phi\,(R) = \{t \mid t \in R \wedge t = (x_1, \ldots, x_k) \wedge \phi\}$, where ϕ is a comparison of the form $x_i = x_j$ or $x_i = o_j$;
- $\mathit{Project}_{i_1 \ldots i_l}$, defined by $\mathit{Project}_{i_1 \ldots i_l}\,(R) = \{t' \mid t \in R \wedge t = (x_1, \ldots, x_k) \wedge t' = (x_{i_1}, \ldots, x_{i_l})\}$.

Intersection can be expressed in terms of the other operators, but we include it here anyway because of its simplicity.

Every operator is coded as a λ-term that takes one or two relations in the list-of-tuples format described in Definition 2.1 as input and produces another relation in the same format as output. The terms do not place any constraints on the type variable τ occurring free in their input and output types, so the output of one term can be used as input for another. Thus, arbitrary relational algebra expressions can be coded by nesting the λ-terms corresponding to the individual operators.

Equal: As a first example, we present a fairly simple term Equal_k that tests two k-tuples $\lambda f.\, f\, o_{i_1} \ldots o_{i_k}$ and $\lambda f.\, f\, o_{j_1} \ldots o_{j_k}$ for equality. The result of the comparison is a Bool, i.e., the term $\lambda u.\, \lambda v.\, u$ if the comparison comes out true and $\lambda u.\, \lambda v.\, v$ otherwise. The code depends on the arity of the tuples involved, so we use the subscript k to indicate that this particular term works for k-tuples.

$$
\begin{array}{l}
\mathit{Equal}_k \colon \mathsf{o}^k \to \mathsf{o}^k \to \mathsf{Bool} \equiv \\
\quad \lambda t \colon \mathsf{o}^k.\, \lambda t' \colon \mathsf{o}^k. \\
\quad\quad \lambda u \colon \tau.\, \lambda v \colon \tau. \\
\quad\quad\quad t\, (\lambda x_1 \colon \mathsf{o} \ldots \lambda x_k \colon \mathsf{o}. \\
\quad\quad\quad\quad t'\, (\lambda y_1 \colon \mathsf{o} \ldots \lambda y_k \colon \mathsf{o}. \\
\quad\quad\quad\quad\quad (\mathit{Eq}\, x_1\, y_1 \\
\quad\quad\quad\quad\quad\quad (\mathit{Eq}\, x_2\, y_2 \\
\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{Eq}\, x_k\, y_k\, u\, v) \ldots v)\, v)))
\end{array}
$$

Here, Eq denotes the equality predicate for constants, of type $\mathsf{o} \to \mathsf{o} \to \mathsf{Bool}$. Once t and t' are instantiated with tuples $\lambda f.\, f\, o_{i_1} \ldots o_{i_k}$ and $\lambda f.\, f\, o_{j_1} \ldots o_{j_k}$, all $(\mathit{Eq}\, x_i\, y_i)$ terms reduce to either $\mathit{True} \equiv \lambda u.\, \lambda v.\, u$ or $\mathit{False} \equiv \lambda u.\, \lambda v.\, v$. Hence the "body"

$$
\begin{array}{l}
\lambda u \colon \tau.\, \lambda v \colon \tau. \\
\quad t\, (\lambda x_1 \colon \mathsf{o} \ldots \lambda x_k \colon \mathsf{o}. \\
\quad\quad t'\, (\lambda y_1 \colon \mathsf{o} \ldots \lambda y_k \colon \mathsf{o}. \\
\quad\quad\quad (\mathit{Eq}\, x_1\, y_1 \\
\quad\quad\quad\quad (\mathit{Eq}\, x_2\, y_2 \\
\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad (\mathit{Eq}\, x_k\, y_k\, u\, v) \ldots v)\, v)))
\end{array}
$$

of the Equal_k predicate reduces to $\lambda u.\, \lambda v.\, u$ if $o_{i_l} = o_{j_l}$ for all $l \in \{1, \ldots, k\}$ and to $\lambda u.\, \lambda v.\, v$ otherwise.

We adopt the following reduction strategy for terms of the form $(\mathit{Equal}_k\, T\, T')$. We assume that the arguments T and T' are already fully reduced. If either T or T' is not an encoding $\lambda f.\, f\, o_{i_1} \ldots o_{i_k}$ of a tuple, no further reductions are performed (we will see that this never happens when we discuss the reduction strategies for the query terms that use *Equal*). Otherwise, T and T' are substituted into the body of the Equal_k predicate and the leftmost redexes are resolved, leading to the substitution of constants for the variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$. Then, the Eq predicates are evaluated from left to right until the final normal form—either $\lambda u.\, \lambda v.\, u$ or $\lambda u.\, \lambda v.\, v$—is reached. Clearly, this is always the case after at most $O(k)$ reduction steps total.

Member: The following term checks whether a k-tuple $\lambda f.\, f\, o_{i_1} \ldots o_{i_k}$ occurs in a list of k-tuples $\lambda c.\, \lambda n.\, c\, T_1\, (c\, T_2 \ldots (c\, T_m\, n) \ldots)$ by comparing the tuple to every element of the list.

$$
\begin{array}{l}
\mathit{Member}_k \colon \mathsf{o}^k \to \{\mathsf{o}^k\} \to \mathsf{Bool} \equiv \\
\quad \lambda t \colon \mathsf{o}^k.\, \lambda r \colon \{\mathsf{o}^k\}. \\
\qquad \lambda u \colon \tau.\, \lambda v \colon \tau. \\
\qquad\quad r\, (\lambda t' \colon \mathsf{o}^k.\, \lambda p \colon \tau.\, (\mathit{Equal}_k\, t\, t')\, u\, p)\, v
\end{array}
$$

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

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

which in turn reduces to $\lambda u.\, \lambda v.\, v \equiv \mathit{False}$, because both $(\mathit{Equal}_2 \ldots)$ terms evaluate to *False*.

Our reduction strategy for terms of the form $(\mathit{Member}_k\, T\, R)$ is as follows. We assume that T and R are already in normal form. If they are not the encoding of a tuple and a relation, respectively, no further reductions are performed (we will see that this never happens when we discuss the reduction strategies for the query terms that use *Member*). Otherwise, T and R are substituted into the body of the *Member* predicate and a copy of the "loop body" $(\mathit{Equal}_k\, T\, t')\, u\, p$ is evaluated once for every tuple in R, beginning with the last tuple and proceeding backwards to the first. At each step, the current tuple of R is substituted for t' and the result of the previous iteration (initially v) is substituted for p. Since both arguments to *Equal* are now encodings of tuples, the reduction strategy for *Equal* described above applies and produces the result of the comparison in $O(k)$ steps. This result, a Bool, is then applied to u and p in order to select the value to pass on to the next stage.

With this reduction strategy, the final result can be determined in $O(k|R|)$ reduction steps, where $|R|$ is the cardinality of the relation encoded by R. The same bound holds for the size of intermediate terms.

Note that there are also "bad" reduction strategies leading to intermediate terms of exponential size. This happens, for example, if in a term $(\mathit{Member}_k\, T\, R)$ with T coding a tuple and R coding a relation, all λ-redexes are resolved before any *Eq*-redexes.

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

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

$$
\begin{array}{l}
\mathit{Intersection}_k \colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv \\
\quad \lambda r \colon \{\mathsf{o}^k\}.\, \lambda r' \colon \{\mathsf{o}^k\}. \\
\qquad \lambda c \colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n \colon \tau. \\
\qquad\quad r\, (\lambda t \colon \mathsf{o}^k.\, \lambda p \colon \tau.\, (\mathit{Member}_k\, t\, r')\, (c\, t\, p)\, p)\, n
\end{array}
$$

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

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

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

The reduction strategy for terms of the form $(\mathit{Intersection}\, R\, R')$—and, with the obvious modifications, for the next two relational operators as well—is as follows. We assume that R and R'

are already in normal form. Nothing is done if they are not encodings of relations. Otherwise, the normal forms are substituted into the body of the *Intersection* operator and the "loop body" $(\mathit{Member}_k\, t\, R')\,(c\, t\, p)\, p$ is evaluated once for each tuple in R, from last to first, with the current tuple substituted for t and the result of the previous iteration (initially n) substituted for p. Since t and R' hold encodings of a tuple and a relation, respectively, at each stage, the reduction strategy for *Member* outlined above applies and produces the result of each stage in $O\,(k|R'|)$ reduction steps. Hence, the final result can be computed in $O\,(k|R||R'|)$ steps, using intermediate terms of size at most $O\,(k|R||R'|)$.

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

$$
\begin{aligned}
&\mathit{Setminus}_k\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv \\
&\quad \lambda r\colon \{\mathsf{o}^k\}.\, \lambda r'\colon \{\mathsf{o}^k\}. \\
&\qquad \lambda c\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad r\,(\lambda t\colon \mathsf{o}^k.\, \lambda p\colon \tau.\, (\mathit{Member}_k\, t\, r')\, p\, (c\, t\, p))\, n
\end{aligned}
$$

Union: The union of two k-ary relations R and R' is formed by starting with R' and adding those tuples of R that do not occur in R'. This also works like intersection.

$$
\begin{aligned}
&\mathit{Union}_k\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv \\
&\quad \lambda r\colon \{\mathsf{o}^k\}.\, \lambda r'\colon \{\mathsf{o}^k\}. \\
&\qquad \lambda c\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad r\,(\lambda t\colon \mathsf{o}^k.\, \lambda p\colon \tau.\, (\mathit{Member}_k\, t\, r')\, p\, (c\, t\, p))\, (r'\, c\, n)
\end{aligned}
$$

If we knew that R and R' were disjoint or if we allowed duplicates, we could implement union by the simpler term

$$
\begin{aligned}
&\mathit{SimpleUnion}\colon \{\alpha\} \to \{\alpha\} \to \{\alpha\} \equiv \\
&\quad \lambda r\colon \{\alpha\}.\, \lambda r'\colon \{\alpha\}. \\
&\qquad \lambda c\colon \alpha \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad r\, c\, (r'\, c\, n),
\end{aligned}
$$

which merely concatenates R and R'. In the general case, however, we need the more complex *Union* operator to ensure a duplicate-free output.

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

$$
\begin{aligned}
&\mathit{Concat}_{k,l}\colon \mathsf{o}^k \to \mathsf{o}^l \to \mathsf{o}^{k+l} \equiv \\
&\quad \lambda t\colon \mathsf{o}^k.\, \lambda t'\colon \mathsf{o}^l. \\
&\qquad \lambda f\colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau. \\
&\qquad\quad t\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}. \\
&\qquad\qquad t'\,(\lambda y_1\colon \mathsf{o} \ldots \lambda y_l\colon \mathsf{o}. \\
&\qquad\qquad\quad f\, x_1 \ldots x_k\, y_1 \ldots y_l))
\end{aligned}
$$

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

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

$$
\begin{aligned}
&\mathit{Times}_{k,l}\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^l\} \to \{\mathsf{o}^{k+l}\} \equiv \\
&\quad \lambda r\colon \{\mathsf{o}^k\}.\, \lambda r'\colon \{\mathsf{o}^l\}. \\
&\qquad \lambda c\colon \mathsf{o}^{k+l} \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad r\,(\lambda t\colon \mathsf{o}^k.\, \lambda p\colon \tau. \\
&\qquad\qquad r'\,(\lambda t'\colon \mathsf{o}^l.\, \lambda p'\colon \tau. \\
&\qquad\qquad\quad c\,(\mathit{Concat}_{k,l}\, t\, t')\, p')\, p)\, n
\end{aligned}
$$

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

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

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

The reduction strategy for an expression $(\mathit{Times}\, R\, R')$ follows the now familiar pattern. Assuming that the arguments are already fully reduced and are encodings of relations, they are substituted into the body of the operator and the "outer loop body" $R'\,(\lambda t'.\, \lambda p'.\, c\,(\mathit{Concat}_{k,l}\, t\, t')\, p')\, p$ is evaluated once for each tuple in R, from last to first, with t and p replaced by the current tuple and the result of the previous iteration, respectively. Evaluating the "outer loop body" just consists of evaluating the "inner loop body" $c\,(\mathit{Concat}_{k,l}\, t\, t')\, p'$ once for each tuple in R', which eventually leads to an evaluation of $(\mathit{Concat}_{k,l}\, t\, t')$ once for every combination of tuples $t \in R$ and $t' \in R'$. It is easy to see that the entire procedure takes $O\,((k+l)|R||R'|)$ reduction steps and uses $O\,((k+l)|R||R'|)$ space for intermediate terms.

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

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

The reduction strategy for an expression $(\mathit{Select}\, R)$ again consists of substituting the value of R into the body of the operator and then evaluating the "loop body" once for each tuple in R. Reduction sequence length and term size are bounded by $O\,(k|R|)$.

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

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

has the disadvantage that the output may contain duplicate tuples. To fix this, we include the projection $\pi(T)$ of tuple T in the output only if T is the *first* tuple in R whose projection is $\pi(T)$. For that, we use an auxiliary term $(\mathit{First}\, T\, R)$ which reduces to *True* iff, among all tuples in R having the same projection as T, T is the first in sequence:

$$\begin{aligned}
&\mathit{First}_{k;i_1,\ldots,i_l} \colon \mathsf{o}^k \to \{\mathsf{o}^k\} \to \mathsf{Bool} \equiv \\
&\quad \lambda t \colon \mathsf{o}^k.\, \lambda r \colon \{\mathsf{o}^k\}. \\
&\qquad \lambda u \colon \tau.\, \lambda v \colon \tau. \\
&\qquad\quad r\,(\lambda t' \colon \mathsf{o}^k.\, \lambda p \colon \tau. \\
&\qquad\qquad \mathit{Equal}_l\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\, t)\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\, t')\,(\mathit{Equal}_k\, t\, t'\, u\, v)\, p)\, u
\end{aligned}$$

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

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

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

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

and then to $\lambda u.\, \lambda v.\, v \equiv \mathit{False}$. The *First* operator is an example of a term that is not oblivious to the ordering of its input. Using the *First* operator, projection can now be coded as follows:

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

For example, $(\mathit{Project}_{2;1}\, [\langle 1,2\rangle, \langle 1,3\rangle])$ evaluates to

$$\begin{aligned}
&\lambda c.\, \lambda n. \\
&\quad (\mathit{First}_{2;1}\, \langle 1,2\rangle\, [\langle 1,2\rangle, \langle 1,3\rangle]) \\
&\qquad (c\, \langle 1\rangle\, ((\mathit{First}_{2;1}\, \langle 1,3\rangle\, [\langle 1,2\rangle, \langle 1,3\rangle])\, (c\, \langle 1\rangle\, n)\, n)) \\
&\qquad ((\mathit{First}_{2;1}\, \langle 1,3\rangle\, [\langle 1,2\rangle, \langle 1,3\rangle])\, (c\, \langle 1\rangle\, n)\, n)
\end{aligned}$$

and then, following the example above, to $\lambda c.\, \lambda n.\, c\, \langle 1\rangle\, n \equiv [\langle 1\rangle]$. The reduction strategy for *Project* is similar to the one for the other relational operators, reducing the arguments first and then evaluating the "loop body" once for each tuple in R. The number of steps and the amount of space needed is $O(k|R|^2)$.

This completes the coding of relational algebra. It is straightforward to verify that every term given above satisfies the typability, well-formed output, and encoding independence conditions of Definition 2.1 and expresses the desired algebra operation. By nesting these terms, arbitrary relational algebra expressions can be coded. Moreover, when such a nested expression is applied to a set of (encoded) input relations and then reduced "from the inside out" according to the reduction strategies given above for the individual operators, the length of the reduction sequence and the size of intermediate terms are polynomial in the size of the inputs. (In fact, the length of the reduction sequence typically matches the running time of a naive implementation of the operator in question.) Therefore, we have the following theorem.

Theorem 3.1 *Any relational algebra query can be PTIME-embedded into TLC$^=$ in the sense of Definition 2.3.*

Remark 3.2 We close this section with the observation that any expression consisting of the operators *SimpleProject*, *SimpleUnion*, and *Times* does not involve the *Eq* constant. This observation will be useful in the next section, when trying to construct the active domain of a database.

4 Coding Fixpoint Queries

With the machinery of the previous section at our disposal, we can now code arbitrary relational queries as TLC$^=$-terms. The next step is to find a way of iterating such queries in order to compute fixpoints. It suffices to perform a polynomial number of iterations using inflationary semantics to capture all PTIME-computable queries [27, 48].

4.1 Iterating Relational Queries

Intuitively, the solution is very simple—we build a sufficiently long list from a Cartesian product of the input relations and then use that list as an iterator to repeat a relational query polynomially many times. The only difficulty lies in getting the types straight, so that the input can serve both as "data" for a relational algebra query and as a "crank" for iterating that same query.

Here are the details. Let $\phi(r, r_1, \ldots, r_l)$ be a relational algebra expression over relational variables $r, r_1, \ldots, r_l$ of arities $k, k_1, \ldots, k_l$ such that the result of ϕ is again of arity k. We wish to compute the inflationary fixpoint of ϕ with respect to r, i.e.,

$$(\mu_r\, \phi)(r_1, \ldots, r_l) := \bigcup_{i=1}^{\infty} \psi^i(\emptyset) = \psi^{n^k}(\emptyset),$$

where ψ is the mapping $r \mapsto r \cup \phi(r, r_1, \ldots, r_l)$ and n is the size of the active domain. This can be done as follows.

Let Q be the TLC$^=$ encoding of the expression $r \cup \phi(r, r_1, \ldots, r_l)$, with variables $r, r_1, \ldots, r_l$ occurring free in Q. Let A be a TLC$^=$ term which computes the active domain of the input, i.e., the set of constants occurring in the relations encoded by $r_1, \ldots, r_l$. A can be expressed as the union of all columns of $r_1, \ldots, r_l$:

$$\begin{array}{l} A\colon \{\mathsf{o}\} \equiv \\ \quad (\mathit{SimpleUnion}\,(\mathit{SimpleProject}_{k_1;1}\, r_1)\,(\mathit{SimpleUnion}\,(\mathit{SimpleProject}_{k_1;2}\, r_1)\ldots \\ \qquad (\mathit{SimpleUnion}\,(\mathit{SimpleProject}_{k_2;1}\, r_2)\,(\mathit{SimpleUnion}\,(\mathit{Project}_{k_2;2}\, r_2)\ldots \\ \qquad\quad \ldots \\ \qquad\qquad (\mathit{SimpleUnion}\,(\mathit{SimpleProject}_{k_l;1}\, r_l)\,(\mathit{SimpleUnion}\,(\mathit{SimpleProject}_{k_l;2}\, r_l)\ldots) \end{array}$$

We use *SimpleUnion* and *SimpleProject* here, since we do not need duplicate elimination and we want to avoid using *Eq* for reasons explained later. The length of the list computed by A is at least n, the size of the active domain. To obtain an iterator of length at least n^k, we form the k-fold Cartesian product of A with itself:

$$Crank\colon \{\mathsf{o}^k\} \equiv (\overbrace{Times\,A\,(Times\,A\ldots(Times\,A\,A)}^{k \text{ factors}}\ldots))$$

Finally, let $Nil \equiv \lambda c{:}\mathsf{o}^k \to \tau \to \tau.\, \lambda n{:}\tau.\, n$ be the empty relation. A straightforward encoding of $(\mu_r\, \phi)$ in the untyped λ-calculus is then the term

$$Fix_\phi \equiv \lambda r_1 \ldots \lambda r_l.\, Crank\,(\lambda t.\, \lambda r.\, Q)\, Nil.$$

Here, t is some variable not occurring in Q; it serves merely to absorb the k-tuple of constants supplied by *Crank* at each iteration. The evaluation of Fix_ϕ proceeds by first forming the normal form of *Crank* according to the reduction rules for relational operators, and then evaluating Q once for each tuple in *Crank*, from last to first, with R bound to the result of the previous iteration (initially the empty relation). This way, the final result of the fixpoint query is produced in a polynomial number of reduction steps involving terms of polynomial size.

4.2 Fixing the Types using "Type-Laundering"

Unfortunately, Fix_ϕ cannot be typed using simple types. This is due to a type conflict between the occurrences of the r_i's in *Crank* and in Q. Inside Q, occurrences of r_i are typed as usual as $\{\mathsf{o}^{k_i}\}$. However, *Crank* is used to iterate over objects of type $\{\mathsf{o}^k\}$, so the occurrences of r_i inside *Crank* must be typed with $\{\mathsf{o}^k\}$ substituted for τ, i.e., as $\{\mathsf{o}^{k_i}\}\,[\tau{:=}\,\{\mathsf{o}^k\}]$. These two typings cannot be unified, since τ does not unify with $\{\mathsf{o}^k\} \equiv (\mathsf{o}^k \to \tau \to \tau) \to \tau \to \tau$.

The solution is to introduce a "type-laundering" operator $Copy_{k_i,k}$ which takes an encoding of a relation R_i of type $\{\mathsf{o}^{k_i}\}\,[\tau{:=}\,\{\mathsf{o}^k\}]$ and produces another encoding of R_i of type $\{\mathsf{o}^{k_i}\}$. That is, $Copy_{k_i,k}$ computes the identity function on encodings of relations, but forces different types for its input and output. $Copy_{k_i,k}$ can be written as follows:

$$
\begin{aligned}
&Copy_{k_i,k}\colon \{\mathsf{o}^{k_i}\}\,[\tau{:=}\,\{\mathsf{o}^k\}] \to \{\mathsf{o}^{k_i}\} \equiv \\
&\quad \lambda r{:}\{\mathsf{o}^{k_i}\}\,[\tau{:=}\,\{\mathsf{o}^k\}]. \\
&\qquad \lambda c{:}\mathsf{o}^{k_i} \to \tau \to \tau.\, \lambda n{:}\tau. \\
&\qquad\quad r\ \ (\lambda t{:}\mathsf{o}^{k_i}\,[\tau{:=}\,\{\mathsf{o}^k\}].\, \lambda p{:}\{\mathsf{o}^k\}. \\
&\qquad\qquad\quad t\,(\lambda x_1{:}\mathsf{o} \ldots \lambda x_{k_i}{:}\mathsf{o}. \\
&\qquad\qquad\qquad \lambda c'{:}\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\tau. \\
&\qquad\qquad\qquad\quad c\,\langle x_1, \ldots, x_{k_i}\rangle\,(p\,c'\,n'))) \\
&\qquad\quad\ \ (\lambda c'{:}\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\tau.\, n) \\
&\qquad\quad\ \ (\lambda t'{:}\mathsf{o}^k.\, \lambda p'{:}\tau.\, p')\, n
\end{aligned}
$$

This somewhat bewildering term arises in the following way. A straightforward "deep" copy operator that does not worry about types would look like this:

$$
\begin{aligned}
&SimpleCopy_{k_i}\colon \{\mathsf{o}^{k_i}\} \to \{\mathsf{o}^{k_i}\} \equiv \\
&\quad \lambda r{:}\{\mathsf{o}^{k_i}\}.
\end{aligned}
$$

$$
\begin{aligned}
&\lambda c{:}\,\mathsf{o}^{k_i} \to \tau \to \tau.\, \lambda n{:}\,\tau. \\
&\quad r\ \ (\lambda t{:}\,\mathsf{o}^{k_i}.\, \lambda p{:}\,\tau. \\
&\qquad\quad t\,(\lambda x_1{:}\,\mathsf{o} \ldots \lambda x_{k_i}{:}\,\mathsf{o}. \\
&\qquad\qquad c\,\langle x_1, \ldots, x_{k_i}\rangle\, p)) \\
&\qquad n
\end{aligned}
$$

It is easy to see that this term computes the identity function on encodings of relations of type $\{\mathsf{o}^{k_i}\}$. If we now want the type of the input to be $\{\mathsf{o}^{k_i}\}\,[\tau{:}=\{\mathsf{o}^k\}]$, then we must modify those subterms of *SimpleCopy* that are "shipped out" of a tuple or a relation, namely $c\,\langle x_1, \ldots, x_{k_i}\rangle\, p$ and $t\,(\lambda x_1 \ldots \lambda x_{k_i}.\, c\,\langle x_1, \ldots, x_{k_i}\rangle\, p)$, to have type $\{\mathsf{o}^k\}$ instead of τ. The easiest way of doing this is by prepending dummy λ-abstractions $\lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau$ to the term $c\,\langle x_1, \ldots, x_{k_i}\rangle\, p$. These dummy abstractions can be removed at the next stage of the iteration by applying the "incoming value" p to the dummy variables c' and n'. This leads to a "loop body"

$$
\begin{aligned}
&(\lambda t{:}\,\mathsf{o}^{k_i}\,[\tau{:}=\{\mathsf{o}^k\}].\, \lambda p{:}\,\{\mathsf{o}^k\}. \\
&\quad t\,(\lambda x_1{:}\,\mathsf{o} \ldots \lambda x_{k_i}{:}\,\mathsf{o}. \\
&\qquad \lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau. \\
&\qquad\quad c\,\langle x_1, \ldots, x_{k_i}\rangle\,(p\,c'\,n'))),
\end{aligned}
$$

which has indeed the correct type $\mathsf{o}^{k_i}\,[\tau{:}=\{\mathsf{o}^k\}] \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}$. Of course, the initial value for the iteration, which used to be just n, has to be changed as well; it becomes $(\lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau.\, n)$. With these changes, the iterator term in the copy operator becomes

$$
\begin{aligned}
&r\ \ (\lambda t{:}\,\mathsf{o}^{k_i}\,[\tau{:}=\{\mathsf{o}^k\}].\, \lambda p{:}\,\{\mathsf{o}^k\}. \\
&\qquad t\,(\lambda x_1{:}\,\mathsf{o} \ldots \lambda x_{k_i}{:}\,\mathsf{o}. \\
&\qquad\quad \lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau. \\
&\qquad\qquad c\,\langle x_1, \ldots, x_{k_i}\rangle\,(p\,c'\,n'))) \\
&\quad (\lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau.\, n)
\end{aligned}
$$

That almost works: if R is a list

$$\lambda c{:}\, o^{k_i}\,[\tau{:}=\{\mathsf{o}^k\}] \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}.\, \lambda n{:}\,\{\mathsf{o}^k\}.\, c\,T_1\,(c\,T_2 \ldots (c\,T_m\,n) \ldots),$$

then the above term reduces, as an easy induction shows, to

$$\lambda c'{:}\, o^k \to \tau \to \tau.\, \lambda n'{:}\,\tau.\, c\,T_1(c\,T_2 \ldots (c\,T_m\,n) \ldots).$$

What remains is to remove the λ-abstractions on c' and n' by supplying dummy arguments of type $\mathsf{o}^k \to \tau \to \tau$ and τ, respectively, and then to λ-abstract on c and n, which leads to the term

$$
\begin{aligned}
&\lambda c{:}\,\mathsf{o}^{k_i} \to \tau \to \tau.\, \lambda n{:}\,\tau. \\
&\quad r\ \ (\lambda t{:}\,\mathsf{o}^{k_i}\,[\tau{:}=\{\mathsf{o}^k\}].\, \lambda p{:}\,\{\mathsf{o}^k\}. \\
&\qquad\quad t\,(\lambda x_1{:}\,\mathsf{o} \ldots \lambda x_{k_i}{:}\,\mathsf{o}. \\
&\qquad\qquad \lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau. \\
&\qquad\qquad\quad c\,\langle x_1, \ldots, x_{k_i}\rangle\,(p\,c'\,n'))) \\
&\qquad (\lambda c'{:}\,\mathsf{o}^k \to \tau \to \tau.\, \lambda n'{:}\,\tau.\, n) \\
&\qquad (\lambda t'{:}\,\mathsf{o}^k.\, \lambda p'{:}\,\tau.\, p')\, n
\end{aligned}
$$

actually used in the definition of the *Copy* operator above. The reduction strategy for an expression $(Copy\ R)$ is as usual, involving a loop over the tuples in R. The normal form is reached after $O(k|R|)$ steps using $O(k|R|)$ space.

With this "type-laundering" machinery at our hands, we can solve the typing problem for fixpoint queries. We modify our original encoding

$$Fix_\phi \equiv \lambda r_1 \ldots \lambda r_l.\, Crank\,(\lambda t.\, \lambda r.\, Q)\, Nil$$

by replacing every occurrence of r_i in Q by $(Copy_{k_i,k}\, r_i)$, i.e., by writing

$$\begin{aligned}&Fix_\phi \equiv\\ &\quad \lambda r_1{:}\,\{\mathsf{o}^{k_1}\}\,[\tau{:}=\{\mathsf{o}^k\}]\ldots\lambda r_l{:}\,\{\mathsf{o}^{k_l}\}\,[\tau{:}=\{\mathsf{o}^k\}].\\ &\qquad Crank\,(\lambda t{:}\,\mathsf{o}^k\,[\tau{:}=\{\mathsf{o}^k\}].\,\lambda r{:}\,\{\mathsf{o}^k\}.\,Q\,[r_1{:}=(Copy_{k_1,k}\,r_1),\ldots,r_l{:}=(Copy_{k_l,k}\,r_l)])\,Nil.\end{aligned}$$

It is straightforward, though tedious, to verify that this is indeed a well-typed term. There is only one subtle point. In *Crank* we never use *Eq* (by Remark 3.2), so we never have to assign a "contaminated" type $\mathsf{o}\to\mathsf{o}\to\mathsf{Bool}\,[\tau{:}=\{\mathsf{o}^k\}]\equiv\mathsf{o}\to\mathsf{o}\to\{\mathsf{o}^k\}\to\{\mathsf{o}^k\}\to\{\mathsf{o}^k\}$ to *Eq*.

Since $(Copy_{k_i,k}\, R_i)$ and R_i encode the same relation, the semantics of Fix_ϕ are unchanged by this modification. The space and time complexity of the evaluation are also unchanged, because it suffices to evaluate the expressions $(Copy_{k_i,k}\, R_i)$ only once.

We now have the machinery in place to compute inflationary fixpoints of relational algebra queries. By Immerman's and Vardi's characterization of the PTIME queries [27, 48], this gives us the ability the express arbitrary PTIME queries, provided we can encode an ordering relation on the active domain of the input. The following term does this:

$$\begin{aligned}&Precedes{:}\,\mathsf{o}\to\mathsf{o}\to\mathsf{Bool}\equiv\\ &\quad\lambda x{:}\,\mathsf{o}.\,\lambda y{:}\,\mathsf{o}.\\ &\qquad\lambda u{:}\,\tau.\,\lambda v{:}\,\tau.\\ &\qquad\quad A\,(\lambda z{:}\,\mathsf{o}.\,\lambda w{:}\,\tau.\,Eq\,z\,x\,u\,(Eq\,z\,y\,v\,w))\,v,\end{aligned}$$

where A computes the active domain as above, but without duplicates. It is easy to see that $(Precedes\,X\,Y)$ reduces to *True* iff X occurs to the left of Y in the list produced by A. The reduction is carried out by the usual loop, assuming that A, X, and Y are already fully reduced. Of course, if *Precedes* is used inside Q, the "type-laundering" substitutions $[r_i{:}=(Copy_{k_i,k}\,r_i)]$ described above apply to it as well.

We have proved the following theorem:

Theorem 4.1 *Any PTIME database query can be PTIME-embedded into TLC$^=$ in the sense of Definition 2.3.*

4.3 An Example: Transitive Closure

To illustrate the concepts above, let us study the representation and evaluation of the transitive closure query in some detail. We compute the transitive closure of a binary relation e by iterating the relational algebra query

$$r \mapsto e \cup (\pi_{1,4}\,(\sigma_{2=3}\,(r\times r)))$$

starting from the empty relation. Here, $\pi_{1,4}$ denotes projection onto the first and fourth attribute and $\sigma_{2=3}$ denotes selection on equal second and third attributes. According to the previous section, the translation into $\mathrm{TLC}^{=}$ of this fixpoint query is (we omit type annotations):

$$
\begin{aligned}
&TC \equiv \lambda e.\\
&\qquad Crank\\
&\qquad\quad (\lambda t.\, \lambda r.\, Union_2\, (Copy_{2,2}\, e)\, (Project_{4;1,4}\, (Select_{4;2=3}\, (Times_{2,2}\, r\, r))))\\
&\qquad\quad Nil,
\end{aligned}
$$

where

$$Crank \equiv (Times_{1,1}\, A\, A)$$

and

$$A \equiv Union_1\, (Project_{2;1}\, e)\, (Project_{2;2}\, e).$$

Note that, for typing reasons, we should use the operators *SimpleUnion* and *SimpleProject* in A. Without loss of generality, we take the more complex operators to illustrate their reductions and to shorten the total number of iterations of *Crank* in the example.

Assume that E encodes the relation $[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]$ and we want to evaluate $(TC\, E)$. According to the reduction strategy for fixpoint queries, we first have to evaluate the expressions *Crank* and $(Copy_{2,2}\, E)$. For the latter, we have (omitting some intermediate steps)

$$
\begin{aligned}
&(Copy_{2,2}\, E)\\
&\rhd\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 2,3\rangle\\
&\qquad\qquad ((\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 3,4\rangle\\
&\qquad\qquad\quad (\lambda c'.\, \lambda n'.\, n)))\, (\lambda t'.\, \lambda p'.\, p')\, n\\
&\rhd\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 2,3\rangle\\
&\qquad\qquad (\langle 3,4\rangle\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, ((\lambda c'.\, \lambda n'.\, n)\, c'\, n'))))\\
&\qquad\qquad\quad (\lambda t'.\, \lambda p'.\, p')\, n\\
&\rhd\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, t\, (\lambda x.\, \lambda y.\, \lambda c'.\, \lambda n'.\, c\, \langle x,y\rangle\, (p\, c'\, n')))\, \langle 2,3\rangle\\
&\qquad\qquad (\lambda c'.\, \lambda n'.\, c\, \langle 3,4\rangle\, n))\\
&\qquad\qquad\quad (\lambda t'.\, \lambda p'.\, p')\, n\\
&\rhd\ \cdots\\
&\rhd\ \lambda c.\, \lambda n.\\
&\qquad (\lambda c'.\, \lambda n'.\\
&\qquad\quad c\, \langle 1,2\rangle\\
&\qquad\qquad (c\, \langle 2,3\rangle\\
&\qquad\qquad\quad (c\, \langle 3,4\rangle\, n)))\, (\lambda t'.\, \lambda p'.\, p')\, n
\end{aligned}
$$

$$\triangleright\ \lambda c.\, \lambda n.\, c\,\langle 1,2\rangle\,(c\,\langle 2,3\rangle\,(c\,\langle 3,4\rangle\,n))$$

$$\equiv [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle],$$

which is what we expect, since *Copy* affects only the type of an encoding, not its value.

To evaluate *Crank*, we first have to evaluate the term

$$A \equiv \mathit{Union}_1\,(\mathit{Project}_{2;1}\,E)\,(\mathit{Project}_{2;2}\,E).$$

We illustrate the reduction sequence for $(\mathit{Project}_{2;1}\,E)$, according to the reduction strategy for projection:

$$
\begin{aligned}
&(\mathit{Project}_{2;1}\,E)\\
&\triangleright\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 2,3\rangle\\
&\qquad\qquad ((\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 3,4\rangle\,n))\\
&\triangleright\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 2,3\rangle\\
&\qquad\qquad ((\mathit{First}_{2;1}\,\langle 3,4\rangle\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,\langle 3,4\rangle)\,n)\,n))\\
&\triangleright\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 2,3\rangle\\
&\qquad\qquad (\mathit{True}\,(c\,(\mathit{Rearrange}_{2;1}\,\langle 3,4\rangle)\,n)\,n))\\
&\triangleright\ \lambda c.\, \lambda n.\\
&\qquad (\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 1,2\rangle\\
&\qquad\quad ((\lambda t.\, \lambda p.\, (\mathit{First}_{2;1}\,t\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\,(c\,(\mathit{Rearrange}_{2;1}\,t)\,p)\,p)\,\langle 2,3\rangle\\
&\qquad\qquad (c\,\langle 3\rangle\,n))\\
&\triangleright\ \cdots\\
&\triangleright\ \lambda c.\, \lambda n.\, c\,\langle 1\rangle\,(c\,\langle 2\rangle\,(c\,\langle 3\rangle\,n))\\
&\equiv [\langle 1\rangle, \langle 2\rangle, \langle 3\rangle]
\end{aligned}
$$

$(\mathit{Project}_{2;2}\,E)$ reduces to $[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle]$ and thus A reduces as follows:

$$
\begin{aligned}
&A \equiv \mathit{Union}_1\,[\langle 1\rangle, \langle 2\rangle, \langle 3\rangle]\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle]\\
&\quad \triangleright\ \lambda c.\, \lambda n.\\
&\qquad\quad (\lambda t.\, \lambda p.\, (\mathit{Member}_1\,t\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,p\,(c\,t\,p))\,\langle 1\rangle\\
&\qquad\qquad ((\lambda t.\, \lambda p.\, (\mathit{Member}_1\,t\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,p\,(c\,t\,p))\,\langle 2\rangle\\
&\qquad\qquad\quad ((\lambda t.\, \lambda p.\, (\mathit{Member}_1\,t\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,p\,(c\,t\,p))\,\langle 3\rangle\\
&\qquad\qquad\qquad (c\,\langle 2\rangle\,(c\,\langle 3\rangle\,(c\,\langle 4\rangle\,n)))))
\end{aligned}
$$

$$\begin{array}{l}
\rhd \lambda c.\, \lambda n. \\
\quad (\lambda t.\, \lambda p.\, (Member_1\, t\, [\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\, p\, (c\, t\, p))\, \langle 1\rangle \\
\quad\quad ((\lambda t.\, \lambda p.\, (Member_1\, t\, [\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\, p\, (c\, t\, p))\, \langle 2\rangle \\
\quad\quad\quad ((Member_1\, \langle 3\rangle\, [\langle 2\rangle, \langle 3\rangle, \langle 4\rangle]) \\
\quad\quad\quad\quad (c\, \langle 2\rangle\, (c\, \langle 3\rangle\, (c\, \langle 4\rangle\, n))) \\
\quad\quad\quad\quad (c\, \langle 3\rangle\, (c\, \langle 2\rangle\, (c\, \langle 3\rangle\, (c\, \langle 4\rangle\, n))))))
\end{array}$$

$$\begin{array}{l}
\rhd \lambda c.\, \lambda n. \\
\quad (\lambda t.\, \lambda p.\, (Member_1\, t\, [\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\, p\, (c\, t\, p))\, \langle 1\rangle \\
\quad\quad ((\lambda t.\, \lambda p.\, (Member_1\, t\, [\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\, p\, (c\, t\, p))\, \langle 2\rangle \\
\quad\quad\quad\quad (c\, \langle 2\rangle\, (c\, \langle 3\rangle\, (c\, \langle 4\rangle\, n))))
\end{array}$$

$$\rhd \cdots$$

$$\rhd \lambda c.\, \lambda n.\, c\, \langle 1\rangle\, (c\, \langle 2\rangle\, (c\, \langle 3\rangle\, (c\, \langle 4\rangle\, n)))$$

$$\equiv [\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle]$$

We then obtain $Crank \equiv (Times_{1,1}\, A\, A) \rhd [\langle 1,1\rangle, \langle 1,2\rangle, \ldots, \langle 4,4\rangle]$. After these reductions, the fixpoint query becomes

$$\begin{array}{l}
(\lambda t.\, \lambda r.\, Union_2\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]\, (Project_{4;1,4}\, (Select_{4;2=3}\, (Times_{2,2}\, r\, r))))\, \langle 1,1\rangle \\
\quad ((\lambda t.\, \lambda r.\, Union_2\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]\, (Project_{4;1,4}\, (Select_{4;2=3}\, (Times_{2,2}\, r\, r))))\, \langle 1,2\rangle \\
\quad\quad \ldots \\
\quad\quad\quad ((\lambda t.\, \lambda r.\, Union_2\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]\, (Project_{4;1,4}\, (Select_{4;2=3}\, (Times_{2,2}\, r\, r))))\, \langle 4,4\rangle \\
\quad\quad\quad\quad (\lambda c.\, \lambda n.\, n)))
\end{array}$$

This expression is now reduced from the inside out by evaluating the "loop body" sixteen times. Since we have already seen the relational operators in action, we will not trace this reduction, but rather just list the values substituted for r at the beginning of each iteration:

Iteration	Value of r
1	$\lambda c.\, \lambda n.\, n$
2	$\lambda c.\, \lambda n.\, c\, \langle 1,2\rangle\, (c\, \langle 2,3\rangle\, (c\, \langle 3,4\rangle\, n))$
3	$\lambda c.\, \lambda n.\, c\, \langle 1,2\rangle\, (c\, \langle 2,3\rangle\, (c\, \langle 3,4\rangle\, (c\, \langle 1,3\rangle\, (c\, \langle 2,4\rangle\, n))))$
4	$\lambda c.\, \lambda n.\, c\, \langle 1,2\rangle\, (c\, \langle 2,3\rangle\, (c\, \langle 3,4\rangle\, (c\, \langle 1,3\rangle\, (c\, \langle 1,4\rangle\, (c\, \langle 2,4\rangle\, n)))))$
$5-16$	unchanged

Thus, the final normal form of the query is $[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle]$, which is indeed the transitive closure of $[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]$.

5 The Complex Object Algebra

We conclude our tour of relational query languages with an encoding of Abiteboul's and Beeri's complex object algebra [1]. This takes us from manipulating "flat" relations to more complicated data structures, namely arbitrary finite trees built from tuple and set constructors. The algebra under consideration is extremely powerful—in fact it expresses any generic elementary time computation on finite structures. Interestingly, no fixpoint or looping construct is needed; the expressive power stems from the ability to create larger and larger sets using the *Powerset* operator.

5.1 Complex Object Algebra Definition

Let us first describe the salient features of the complex object model and its operator algebra. The presentation here is taken with slight modifications from [2].

5.1.1 Complex Object Databases

We assume the existence of the following countably infinite and pairwise disjoint sets of atomic elements: relation names $\{R_1, R_2, \ldots\}$, attributes $\{A_1, A_2, \ldots\}$, constants $D = \{d_1, d_2, \ldots\}$.

Definition 5.1 *The abstract syntax σ and the interpretation $[\![\sigma]\!]$ of* sorts *are given by:*

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

An element of a sort is called a complex object.

Note that, each complex object can be viewed as a finite tree. A complex object of the form $\langle \ldots \rangle$ (resp., $\{\ldots\}$) is said to be a *tuple* (resp., a *set*). The tuple fields are viewed as unordered. We therefore do not distinguish, for instance, between sorts $\langle A \colon D, B \colon D \rangle$ and $\langle B \colon D, A \colon D \rangle$, or between objects $\langle A \colon 2, B \colon 2 \rangle$ and $\langle B \colon 2, A \colon 2 \rangle$. Note also that (because of the empty set) a complex object may belong to more than one sort.

Definition 5.2 *A* database schema *is a pair $(\mathcal{R}, \mathcal{S})$ where $\mathcal{R}$ is a set of relation names and $\mathcal{S}$ is a function from $\mathcal{R}$ to sorts. A* database instance *$\mathcal{I}$ of a schema $(\mathcal{R}, \mathcal{S})$ is a function from $\mathcal{R}$ such that for each $R \in \mathcal{R}$, $\mathcal{I}(R) \in [\![\{\mathcal{S}(R)\}]\!]$.*

Note that by definition, each $\mathcal{I}(R)$ is a set and can be viewed as a finite tree. Figure 5.1.1 shows an instance $\mathcal{I}$ of $(\{R_1, R_2, R_3\}, \mathcal{S})$ where $\mathcal{S}(R_1) = \mathcal{S}(R_3) = \langle A \colon D, B \colon \{\langle A_1 \colon D, A_2 \colon D \rangle\}\rangle$ and $\mathcal{S}(R_2) = \langle A \colon D, A_1 \colon D, A_2 \colon D \rangle$.

We next define the computable queries using the auxiliary concept of "C-genericity." The queries that we consider are generalizations of the *computable relational queries* of Chandra and Harel [9].

Definition 5.3 *Let C be a finite set of constants. A mapping f from instances to instances is C-*generic *if for each permutation ρ of the constants which is the identity on C and each instance $\mathcal{I}$, $f(\rho(\mathcal{I})) = \rho(f(\mathcal{I}))$. A* computable query *$q$ is a mapping from instances over a schema $(\mathcal{R}, \mathcal{S})$ to instances over a schema $(\mathcal{R}', \mathcal{S}')$ which is (1) a partial recursive function and (2) C_q-generic, where C_q is the set of constants occurring in (the selection predicates of) q. We use the term* sort of a query *for the sort of its output.*

5.1.2 The Algebraic Operators

We next define a many sorted algebra for complex objects. Let $(\mathcal{R}, \mathcal{S})$ be a schema.

Definition 5.4 *For each R in $\mathcal{R}$, $[R]$ is an* atomic (algebraic) query *and* sort$([R]) = \{\mathcal{S}(R)\}$.

<table>
<tr><th>A</th><th>B</th></tr>
<tr><td>d_1</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_2</td></tr><tr><td>d_3</td><td>d_4</td></tr></table></td></tr>
<tr><td>d_1</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_3</td><td>d_4</td></tr><tr><td>d_5</td><td>d_6</td></tr></table></td></tr>
<tr><td>d_2</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_3</td></tr><tr><td>d_2</td><td>d_4</td></tr></table></td></tr>
</table>

$\mathcal{I}(R_1)$

A	A_1	A_2
d_1	d_1	d_2
d_1	d_3	d_4
d_1	d_5	d_6
d_2	d_1	d_3
d_2	d_2	d_4

$\mathcal{I}(R_2)$

<table>
<tr><th>A</th><th>B</th></tr>
<tr><td>d_1</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_2</td></tr><tr><td>d_3</td><td>d_4</td></tr><tr><td>d_5</td><td>d_6</td></tr></table></td></tr>
<tr><td>d_2</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_3</td></tr><tr><td>d_2</td><td>d_4</td></tr></table></td></tr>
</table>

$\mathcal{I}(R_3)$

Figure 1: A database instance.

Note again that because of the empty set, an atomic query may be of more than one sort. To simplify the presentation, we ignore this aspect here and assume in the following that each query q has a unique sort.

More elaborate queries are obtained by combining atomic queries via various algebra operators:

Definition 5.5 *A* composite (algebraic) query *is one of the following:*

- **Set operations:**
  - *If* q_1 *and* q_2 *are queries with* $\mathrm{sort}(q_1) = \mathrm{sort}(q_2)$*, then* $q_1 \cap q_2$*,* $q_1 \cup q_2$*,* $q_1 - q_2$*, and* $Powerset(q_1)$ *are queries and* $\mathrm{sort}(q_1 \cap q_2) = \mathrm{sort}(q_1 \cup q_2) = \mathrm{sort}(q_1 - q_2) = \mathrm{sort}(q_1)$*,* $\mathrm{sort}(Powerset(q_1)) = \{\mathrm{sort}(q_1)\}$*.*
  - *If* q *is of sort* $\{\sigma\}$*, then* $Set(q)$ *is a query of sort* $\{\{\sigma\}\}$*.*
  - *If* q *is of sort* $\{\{\sigma\}\}$*, then* $Setcomb(q)$ *is a query of sort* $\{\sigma\}$*.*
- **Tuple operations:** *If* q *is a query of sort* $\{\langle B_1{:}\sigma_1, \ldots, B_j{:}\sigma_j\rangle\}$*, then*
  - $Select_\phi(q)$ *is a query of sort* $\{\langle B_1{:}\sigma_1, \ldots, B_j{:}\sigma_j\rangle\}$*; the selector* ϕ *is (with obvious restrictions on sorts) of the form:* $B_1 = d$*,* $B_1 = B_2$*,* $B_1 \in B_2$ *or* $B_1 = B_2.C$*, where* d *is a constant and in the last case it is required that* σ_2 *is a tuple sort with a* C*-field;*
  - $Project_{B_1,\ldots,B_k}(q)$ *is a query of sort* $\{\langle B_1{:}\sigma_1, \ldots, B_k{:}\sigma_k\rangle\}$*.*
  - *If* q *is of sort* $\{\sigma\}$*, then* $Tup_A(q)$ *is a query of sort* $\{\langle A{:}\sigma\rangle\}$*.*
  - *If* q *is of sort* $\{\langle A{:}\sigma\rangle\}$*, then* $Tupcomb(q)$ *is a query of sort* $\{\sigma\}$*.*
- **Cartesian product:** *If* q_i*,* $i = 1 \ldots m$*, are queries with sorts* $\sigma_i = \{\left\langle B^i_1{:}\sigma^i_1, \ldots, B^i_{j_i}{:}\sigma^i_{j_i}\right\rangle\}$ *and the attribute sets are disjoint, then* $q_1 \times \ldots \times q_m$ *is a query with* $\mathrm{sort}(q_1 \times \cdots \times q_m) = \{\left\langle B^1_1{:}\sigma^1_1, \ldots, B^1_{j_1}{:}\sigma^1_{j_1}, \ldots, B^m_1{:}\sigma^m_1, \ldots, B^m_{j_m}{:}\sigma^m_{j_m}\right\rangle\}$*.*

Definition 5.6 *The* answer *to a query q on an instance $\mathcal{I}$, denoted by $q(\mathcal{I})$, is defined as follows:*

- $[R](\mathcal{I}) = \mathcal{I}(R)$;
- $(q_1 \cap q_2)(\mathcal{I}) = q_1(\mathcal{I}) \cap q_2(\mathcal{I})$ *and similarly for* $\cup$, $-$, *Powerset;*
- $Set(q_1)(\mathcal{I}) = \{\{t\} \mid t \in q_1(\mathcal{I})\}$;
- $Setcomb(q_1)(\mathcal{I}) = \bigcup q_1(\mathcal{I})$;
- $Select_\phi(q_1)(\mathcal{I}) = \{v \mid v \in q_1(\mathcal{I}), v \vdash \phi\}$, *where* $\vdash$ *is defined by:*

  $\langle B_1 : v_1, \ldots\rangle \vdash B_1 = d$ *if* $v_1 = d$,
  $\langle B_1 : v_1, \ldots\rangle \vdash B_1 = B_2$ *if* $v_1 = v_2$,
  $\langle B_1 : v_1, \ldots\rangle \vdash B_1 \in B_2$ *if* $v_1 \in v_2$,
  $\langle B_1 : v_1, \ldots\rangle \vdash B_1 = B_2.C$ *if* $v_2 = \langle C : v_1, \ldots\rangle$;

- $Project_{B_1,\ldots,B_k}(q_1)(\mathcal{I}) = \{\langle B_1 : v_1, \ldots, B_k : v_k\rangle \mid \exists\, v_{k+1}, \ldots, v_j : \langle B_1 : v_1, \ldots, B_j : v_j\rangle \in q_1(\mathcal{I})\}$;
- $Tup_A(q_1(\mathcal{I})) = \{\langle A : t\rangle \mid t \in q_1(\mathcal{I})\}$;
- $Tupcomb(q_1)(\mathcal{I}) = \{v_1, \ldots, v_j\}$ *if* $q_1(\mathcal{I}) = \{\langle A : v_1\rangle, \ldots, \langle A : v_j\rangle\}$;
- *Let* q_i, $i = 1 \ldots m$, *be queries of sorts* $\sigma_i = \{\left\langle B^i_1 : \sigma^i_1, \ldots, B^i_{j_i} : \sigma^i_{j_i}\right\rangle\}$. *Then* $q_1 \times \cdots \times q_m(\mathcal{I}) = \{\left\langle B^1_1 : x^1_1, \ldots, B^1_{j_1} : x^1_{j_1}, \ldots, B^m_1 : x^m_1, \ldots, B^m_{j_m} : x^m_{j_m}\right\rangle \mid \forall\, i : \left\langle B^i_1 : x^i_1, \ldots, B^i_{j_i} : x^i_{j_i}\right\rangle \in q_i(\mathcal{I})\}$.

This algebra we call ALG. For the algebra without powerset operation we use ALG$^-$.

5.1.3 The Expressive Power of ALG and ALG$^-$

A query is an *elementary query* if it is a computable query and has elementary-recursive data complexity[2] with respect to the database size. It turns out that a query is in ALG/CALC iff it is an elementary query (we refer to [26, 31] for detailed definitions and proofs). Furthermore, Hull and Su exhibited a hierarchy of classes of queries based on the level of set nesting allowed in temporary predicates [26]. One level leads to SO, i.e., relational calculus extended with second order quantification [10]. Kuper and Vardi showed a somewhat analogous result [31]: there exists a hierarchy of classes of powerset algebras based on the allowable level of nesting of powersets.

One might wonder what is at the bottom of the hierarchy, i.e., what is the power of the algebra *without* the powerset operation. One can show that queries in ALG$^-$ can be computed in PTIME with respect to the size of the database. The (nested) use of the powerset operation is solely responsible for (the stack of) exponential blowups. Thus ALG$^-$ is strictly less expressive than ALG. Paredaens and Van Gucht showed that a query from a relational schema to a relational schema is expressible in ALG$^-$ iff it is expressible in the classical relational algebra [39]. A restricted calculus is presented in [1]. The restrictions consist of syntactically forcing formulas to provide limited ranges to all variables. The restricted calculus is called CALC$^-$. The equivalence of ALG$^-$ and CALC$^-$ is shown in [1].

[2]We are concerned here exclusively with the *data* complexity in the sense of [10, 48].

5.2 Coding ALG$^-$.

We begin by extending our embedding of relational algebra to deal with complex objects and to express the *Tup*, *Tupcomb*, *Set*, and *Setcomb* operators. The *Powerset* operator comes with its own share of difficulties and its treatment is postponed to the next section.

In the discussion below, we assume for simplicity that the domain over which all complex objects are defined is the set $\mathcal{D} = \{o_1, o_2, \ldots\}$ of TLC$^=$ constants. Furthermore, we regard the components of a tuple as *ordered* and rely on their relative position instead of attribute names for identification.

We encode complex objects in the natural way, using TLC$^=$ constants at the bottom level and then combining them into tuples and lists. Relations in the complex object model are just sets of objects of a common sort, and they are encoded as a TLC$^=$ list of the encodings of their elements.

More precisely, we have the following encoding convention:

Definition 5.7 *Let X be a complex object. Then an* encoding *of X, denoted by $\overline{X}$, is a TLC$^=$ term E such that*

- $E = o_j$, *if X is an atomic element o_j,*
- $E = \lambda f.\, f\,\overline{X_1}\ldots\overline{X_k}$, *if X is a tuple $\langle X_1, \ldots, X_k\rangle$,*
- $E = \lambda c.\,\lambda n.\,(c\,\overline{X_1}\,(c\,\overline{X_2}\ldots(c\,\overline{X_k}\,n)\ldots))$, *if X is a set $\{X_1, \ldots, X_k\}$.*

If $(R_1{:}\,\sigma_1, \ldots, R_l{:}\,\sigma_l)$ is a database instance, then an encoding of R_i, denoted by $\overline{R_i}$, is an encoding of the complex object $\{X \mid X \in R_i\}$. Query terms *are defined as in Definition 2.1, with arities replaced by sorts.*

Note that the type of $\overline{X}$ corresponds to the sort of X: The encoding of an atomic element has type o, the encoding of an object of sort $\langle\alpha_1, \ldots, \alpha_k\rangle$ has type $\alpha'_1 \times \cdots \times \alpha'_k \equiv (\alpha'_1 \to \cdots \to \alpha'_k \to \tau) \to \tau$, where α'_i is the type corresponding to sort α_i, and the encoding of an object of sort $\{\alpha\}$ has type $\{\alpha'\} \equiv (\alpha' \to \tau \to \tau) \to \tau \to \tau$. Because of this tight correspondence, we often simply speak of the "type" of a complex object if the distinction between its sort and the type of its encoding does not matter.

Most of the relational algebra operators described in Section 3 work with little or no change in the complex object setting. However, the *Equal* operator becomes more complicated, because it has to implement a "deep" comparison that traverses the structure of two complex objects and checks equality at each level. Because set equality involves subset testing, which in turn is defined in terms of the membership predicate, we have to define the *Equal*, *Subset*, and *Member* predicates simultaneously, using structural induction over the argument types. Thus, for each complex object type α, we define below three terms $Equal_\alpha$, $Member_\alpha$, and $Subset_{\{\alpha\}}$ that encode the corresponding predicate for objects of that type, using the already defined encodings for objects of less complex type.

$$
\begin{aligned}
&Equal_{\mathsf{o}}{:}\,\mathsf{o} \to \mathsf{o} \to \mathsf{Bool} \equiv\\
&\quad \lambda x{:}\,\mathsf{o}.\,\lambda y{:}\,\mathsf{o}.\,Eq\,x\,y
\end{aligned}
$$

$$
\begin{aligned}
&Equal_{\{\alpha\}}{:}\,\{\alpha\} \to \{\alpha\} \to \mathsf{Bool} \equiv\\
&\quad \lambda s{:}\,\{\alpha\}.\,\lambda s'{:}\,\{\alpha\}.\\
&\quad\quad \lambda u{:}\,\tau.\,\lambda v{:}\,\tau.\\
&\quad\quad\quad (Subset_{\{\alpha\}}\,s\,s')\,((Subset_{\{\alpha\}}\,s'\,s)\,u\,v)\,v
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{Equal}_{\alpha_1\times\cdots\times\alpha_k}\colon \alpha_1\times\cdots\times\alpha_k\to\alpha_1\times\cdots\times\alpha_k\to\mathsf{Bool}\equiv\\
\quad \lambda t\colon \alpha_1\times\cdots\times\alpha_k.\,\lambda t'\colon \alpha_1\times\cdots\times\alpha_k.\\
\quad\quad \lambda u\colon \tau.\,\lambda v\colon \tau.\\
\quad\quad\quad t\,(\lambda x_1\colon \alpha_1\ldots\lambda x_k\colon \alpha_k.\\
\quad\quad\quad\quad t'\,(\lambda y_1\colon \alpha_1\ldots\lambda y_k\colon \alpha_k.\\
\quad\quad\quad\quad\quad ((\mathit{Equal}_{\alpha_1}\,x_1\,y_1)\\
\quad\quad\quad\quad\quad\quad \ldots\\
\quad\quad\quad\quad\quad\quad\quad ((\mathit{Equal}_{\alpha_k}\,x_k\,y_k)\,u\,v)\,v)\ldots v)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Member}_{\alpha}\colon \alpha\to\{\alpha\}\to\mathsf{Bool}\equiv\\
\quad \lambda x\colon \alpha.\,\lambda s\colon \{\alpha\}.\\
\quad\quad \lambda u\colon \tau.\,\lambda v\colon \tau.\\
\quad\quad\quad s\,(\lambda y\colon \alpha.\,\lambda p\colon \tau.\,(\mathit{Equal}_{\alpha}\,x\,y)\,u\,p)\,v
\end{array}
$$

$$
\begin{array}{l}
\mathit{Subset}_{\{\alpha\}}\colon \{\alpha\}\to\{\alpha\}\to\mathsf{Bool}\equiv\\
\quad \lambda s\colon \{\alpha\}.\,\lambda s'\colon \{\alpha\}.\\
\quad\quad \lambda u\colon \tau.\,\lambda v\colon \tau.\\
\quad\quad\quad s\,(\lambda x\colon \alpha.\,\lambda p\colon \tau.\,(\mathit{Member}_{\alpha}\,x\,s')\,p\,v)\,u
\end{array}
$$

The reduction strategies for these operators are as follows. We assume that the arguments are already in normal form and are encodings of complex objects of the required type. First, the arguments are substituted into the operator body. Then, for a term $(\mathit{Equal}_{\{\alpha\}}\,S\,S')$, reduction proceeds by reducing the two *Subset* terms to either *True* or *False* according to the rules below and then picking either u or v as appropriate. For a term $(\mathit{Equal}_{\alpha_1\times\cdots\times\alpha_k}\,T\,T')$, reduction of the body proceeds by instantiating $x_1,\ldots,x_k$ and $y_1,\ldots,y_k$ and reducing the $\mathit{Equal}_{\alpha_i}$ tests in sequence until the normal form is produced. For terms $(\mathit{Member}_{\alpha}\,X\,S)$, reduction of the body involves a loop over the elements of S in reverse sequence, evaluating the Equal_{α} predicate at each step and passing either u or the result of the previous iteration on to the next stage. The strategy for terms $(\mathit{Subset}_{\{\alpha\}}\,S\,S')$ is analogous. A straightforward induction shows that for each operator, the reduction sequence length and intermediate term size are bounded by the product of the argument sizes.

The main change required to make the relational operators of Section 3 work for complex objects is to use the new version of *Equal* and *Member*. Also, there will now typically be one operator for each complex object sort instead of each relation arity. For example, the code for *Intersection* becomes:

$$
\begin{array}{l}
\mathit{Intersection}_{\{\alpha\}}\colon \{\alpha\}\to\{\alpha\}\to\{\alpha\}\equiv\\
\quad \lambda s\colon \{\alpha\}.\,\lambda s'\colon \{\alpha\}.\\
\quad\quad \lambda c\colon \alpha\to\tau\to\tau.\,\lambda n\colon \tau.\\
\quad\quad\quad s\,(\lambda x\colon \alpha.\,\lambda p\colon \tau.\,(\mathit{Member}_{\alpha}\,x\,s')\,(c\,x\,p)\,p)\,n
\end{array}
$$

The code for *Union* and *Setminus* is similar, and the reduction strategies for these operators are the same as in Section 3.

The *Times* operator for complex objects is exactly the same as the one for relations, except that it is typed differently. If R and R' are encodings of relations of type $\{\alpha_1 \times \cdots \times \alpha_k\}$ and $\{\beta_1 \times \cdots \times \beta_l\}$, respectively, then their Cartesian product is computed by

$$
\begin{aligned}
&\mathit{Times}_{k,l}\colon \{\alpha_1 \times \cdots \times \alpha_k\} \to \{\beta_1 \times \cdots \times \beta_l\} \to \{\alpha_1 \times \cdots \times \alpha_k \times \beta_1 \times \cdots \times \beta_l\} \equiv \\
&\quad \lambda r\colon \{\alpha_1 \times \cdots \times \alpha_k\}.\, \lambda r'\colon \{\beta_1 \times \cdots \times \beta_l\}. \\
&\qquad \lambda c\colon \alpha_1 \times \cdots \times \beta_l \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad r\,(\lambda t\colon \alpha_1 \times \cdots \times \alpha_k.\, \lambda p\colon \tau. \\
&\qquad\qquad r'\,(\lambda t'\colon \beta_1 \times \cdots \times \beta_l.\, \lambda p'\colon \tau. \\
&\qquad\qquad\quad c\,(\mathit{Concat}_{k,l}\, t\, t')\, p')\, p)\, n,
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{Concat}_{k,l}\colon \alpha_1 \times \cdots \times \alpha_k \to \beta_1 \times \cdots \times \beta_l \to \alpha_1 \times \cdots \times \alpha_k \times \beta_1 \times \cdots \times \beta_l \equiv \\
&\quad \lambda t\colon \alpha_1 \times \cdots \times \alpha_k.\, \lambda t'\colon \beta_1 \times \cdots \times \beta_l. \\
&\qquad \lambda f\colon \alpha_1 \to \cdots \to \beta_l \to \tau. \\
&\qquad\quad t\,(\lambda x_1\colon \alpha_1 \ldots \lambda x_k\colon \alpha_k. \\
&\qquad\qquad t'\,(\lambda y_1\colon \beta_1 \ldots \lambda y_l\colon \beta_l. \\
&\qquad\qquad\quad f\, x_1 \ldots x_k\, y_1 \ldots y_l)).
\end{aligned}
$$

Note that the code for these operators depends only on k and l, but not on the exact nature of the α_i's and β_j's. The reduction strategy for *Times* is the same as in Section 3.

For *Project*, it again suffices to use the new version of *Equal* instead of the old one. If R is an encoding of a relation of type $\{\alpha_1 \times \cdots \times \alpha_k\}$, the projection of R onto columns $i_1, \ldots, i_l$ is computed by

$$
\begin{aligned}
&\mathit{Project}_{\alpha_1 \times \cdots \times \alpha_k; i_1, \ldots, i_l}\colon \{\alpha_1 \times \cdots \times \alpha_k\} \to \{\alpha_{i_1} \times \cdots \times \alpha_{i_l}\} \equiv \\
&\quad \lambda r\colon \{\alpha_1 \times \cdots \times \alpha_k\}. \\
&\qquad \lambda c\colon \alpha_{i_1} \times \cdots \times \alpha_{i_l} \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad r\,(\lambda t\colon \alpha_1 \times \cdots \times \alpha_k.\, \lambda p\colon \tau.\, (\mathit{First}_{\alpha_1 \times \cdots \times \alpha_k; i_1, \ldots, i_l}\, t\, r)\, (c\,(\mathit{Rearrange}_{k; i_1, \ldots, i_l}\, t)\, p)\, p)\, n,
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{First}_{\alpha_1 \times \cdots \times \alpha_k; i_1, \ldots, i_l}\colon \alpha_1 \times \cdots \times \alpha_k \to \{\alpha_1 \times \cdots \times \alpha_k\} \to \mathsf{Bool} \equiv \\
&\quad \lambda t\colon \alpha_1 \times \cdots \times \alpha_k.\, \lambda r\colon \{\alpha_1 \times \cdots \times \alpha_k\}. \\
&\qquad \lambda u\colon \tau.\, \lambda v\colon \tau. \\
&\qquad\quad r\;\; (\lambda t'\colon \alpha_1 \times \cdots \times \alpha_k.\, \lambda p\colon \tau. \\
&\qquad\qquad\quad \mathit{Equal}_{\alpha_{i_1} \times \cdots \times \alpha_{i_l}}\, (\mathit{Rearrange}_{k; i_1, \ldots, i_l}\, t)\, (\mathit{Rearrange}_{k; i_1, \ldots, i_l}\, t') \\
&\qquad\qquad\qquad (\mathit{Equal}_{\alpha_1 \times \cdots \times \alpha_k}\, t\, t'\, u\, v) \\
&\qquad\qquad\qquad p) \\
&\qquad\qquad u
\end{aligned}
$$

and $\mathit{Rearrange}_{k; i_1, \ldots, i_l}$ is defined as in Section 3. The reduction strategy for *Project* remains unchanged.

The *Select* operator has to cater to a larger set of selection conditions than in relational algebra, but its basic structure remains unchanged:

$$\begin{array}{l} \mathit{Select}_{\{\alpha_1\times\cdots\times\alpha_k\};\phi}\colon \{\alpha_1 \times \cdots \times \alpha_k\} \to \{\alpha_1 \times \cdots \times \alpha_k\} \equiv \\ \quad \lambda r\colon \{\alpha_1 \times \cdots \times \alpha_k\}. \\ \quad\quad \lambda c\colon \alpha_1 \times \cdots \times \alpha_k \to \tau \to \tau.\, \lambda n\colon \tau. \\ \quad\quad\quad r\,(\lambda t\colon \alpha_1 \times \cdots \times \alpha_k.\, \lambda p\colon \tau. \\ \quad\quad\quad\quad t\,(\lambda x_1\colon \alpha_1 \ldots \lambda x_k\colon \alpha_k.\, (\phi\, x_1 \ldots x_k)\,(c\, t\, p)\, p))\, n, \end{array}$$

where $\phi\colon \alpha_1 \to \cdots \to \alpha_k \to \mathsf{Bool}$ encodes the selection condition. For example, to select on the condition $x_1 = o_1$, ϕ would be written as $\lambda x_1 \ldots \lambda x_k.\, \mathit{Eq}\, x_1\, o_1$; to select on $x_1 \in x_2$, ϕ would be written as $\lambda x_1 \ldots \lambda x_k.\, \mathit{Member}\, x_1\, x_2$, and so forth. The reduction strategy for *Select* is the same as in Section 3.

The new operators *Tup*, *Tupcomb*, and *Set* can be implemented in a straightforward fashion (remember that α^1 is an abbreviation for the type $(\alpha \to \tau) \to \tau$):

$$\begin{array}{l} \mathit{Tup}\colon \{\alpha\} \to \{\alpha^1\} \equiv \\ \quad \lambda s\colon \{\alpha\}. \\ \quad\quad \lambda c\colon \alpha^1 \to \tau \to \tau.\, \lambda n\colon \tau. \\ \quad\quad\quad s\,(\lambda x\colon \alpha.\, \lambda p\colon \tau.\, c\,(\lambda f\colon \alpha \to \tau.\, f\, x)\, p)\, n \end{array}$$

$$\begin{array}{l} \mathit{Tupcomb}\colon \{\alpha^1\} \to \{\alpha\} \equiv \\ \quad \lambda r\colon \{\alpha^1\}. \\ \quad\quad \lambda c\colon \alpha \to \tau \to \tau.\, \lambda n\colon \tau. \\ \quad\quad\quad r\,(\lambda t\colon \alpha^1.\, \lambda p\colon \tau.\, t\,(\lambda x\colon \alpha.\, c\, x\, p))\, n \end{array}$$

$$\begin{array}{l} \mathit{Set}\colon \{\alpha\} \to \{\{\alpha\}\} \equiv \\ \quad \lambda s\colon \{\alpha\}. \\ \quad\quad \lambda c\colon \{\alpha\} \to \tau \to \tau.\, \lambda n\colon \tau. \\ \quad\quad\quad s\,(\lambda x\colon \alpha.\, \lambda p\colon \tau.\, c\,(\lambda c'\colon \alpha \to \tau \to \tau.\, \lambda n'\colon \tau.\, c'\, x\, n')\, p)\, n \end{array}$$

As long as the arguments are reduced to fully instantiated encodings of complex objects first, any reduction strategy will work in polynomial time and space for these terms.

The *Setcomb* operator, which "flattens" a set of sets by computing the union of all its members, is slightly more involved because it has to deal with duplicate elimination. The technique is basically the same as in the *Project* operator, involving a predicate $(\mathit{First}\, X\, S'\, S)$ that evaluates to *True* iff $X \in S' \in S$ and S' is the first element of S that contains X. Using this predicate, $(\mathit{Setcomb}\, S)$ can be computed duplicate-free by looping over all elements $S' \in S$ and for each S', including only those $X \in S'$ in the output for which $(\mathit{First}\, X\, S'\, S)$ evaluates to *True*. That is exactly what the following term does.

$$\begin{array}{l} \mathit{Setcomb}\colon \{\{\alpha\}\} \to \{\alpha\} \equiv \\ \quad \lambda s\colon \{\{\alpha\}\}. \\ \quad\quad \lambda c\colon \alpha \to \tau \to \tau.\, \lambda n\colon \tau. \\ \quad\quad\quad s\,(\lambda s'\colon \{\alpha\}.\, \lambda p\colon \tau. \\ \quad\quad\quad\quad s'\,(\lambda x\colon \alpha.\, \lambda p'\colon \tau.\, (\mathit{First}_\alpha\, x\, s'\, s)\,(c\, x\, p')\, p')\, p)\, n, \end{array}$$

where

$$\begin{array}{l} First_\alpha \colon \alpha \to \{\alpha\} \to \{\{\alpha\}\} \to \mathsf{Bool} \equiv \\ \quad \lambda x\colon \alpha.\, \lambda s'\colon \{\alpha\}.\, \lambda s\colon \{\{\alpha\}\}. \\ \qquad \lambda u\colon \tau.\, \lambda v\colon \tau. \\ \qquad\quad s\, (\lambda s''\colon \{\alpha\}.\, \lambda p\colon \tau. \\ \qquad\qquad s''\, (\lambda y\colon \alpha.\, \lambda p'\colon \tau.\, Equal_\alpha\; x\; y\, (Equal_{\{\alpha\}}\; s'\; s''\; u\; v)\, p')\, p)\, v. \end{array}$$

The reduction strategy for an expression $(Setcomb\, S)$ operator is as follows. We assume that S is in normal form and is an encoding of a complex object of sort set of sets. S is substituted into the body of the operator and the "outer loop body" $s'\,(\lambda x.\, \lambda p'.\, (First\; x\; s'\; S)\,(c\; x\; p')\; p')\; p$ is evaluated once for each set in S, from last to first, with s' and p replaced by the current set and the result of the previous iteration stage, respectively. The evaluation of the "outer loop body" is done by evaluating the "inner loop body" $(First\; x\; s'\; S)\,(c\; x\; p')\; p'$ once for each element of the current set, from last to first, with x and p' replaced by the current element and the result of the previous "inner" iteration stage, respectively. The evaluation of the "inner loop body" in turn is done by first reducing the expression $(First\; x\; s'\; S)$, with the proper values substituted for x and s', to either *True* or *False* and then returning either $(c\; x\; p')$ or p' as appropriate. The reduction strategy for the *First* predicate is essentially the same, except that the evaluation of the "inner loop body" involves reducing two *Equal* terms, which is done according to the reduction strategies specified for *Equal*. It is easy to see that the entire procedure uses time and space polynomial in the size of R.

This completes the coding of ALG$^-$. It is straightforward to verify that every term given above satisfies the typability, well-formed output, and encoding independence conditions of Definition 2.1 and expresses the desired algebra operation. By nesting these terms, arbitrary ALG$^-$ expressions can be coded. Moreover, when such a nested expression is applied to a set of (encoded) input objects and then reduced "from the inside out" according to the reduction strategies given above for the individual operators, the length of the reduction sequence and the size of intermediate terms are polynomial in the size of the inputs. Therefore, we have the following theorem.

Theorem 5.8 *Any ALG$^-$ query can be PTIME-embedded into TLC$^=$ in the sense of Definition 2.3.*

5.3 Coding Powerset

The one remaining operator, *Powerset*, accounts for the great expressiveness of the algebra, but poses a variety of difficult typing problems. Writing a λ-term that computes powersets is not too hard—essentially a generalization of the well-known term for exponentiating Church numerals. However, getting the types straight, so that the *Powerset* operator can be used in conjunction with the other algebra operators, is somewhat tricky.

Let us first look at a simple version of *Powerset* that is not concerned with types. Its operation is guided by the inductive definition of the powerset $\mathcal{P}(S)$ of a set S:

$$\begin{array}{rcl} \mathcal{P}(\emptyset) & = & \{\emptyset\} \\ \mathcal{P}(\{x_1, \ldots, x_n\}) & = & \mathcal{P}(\{x_1, \ldots, x_{n-1}\}) \cup \{\{x_n\} \cup s \mid s \in \mathcal{P}(\{x_1, \ldots, x_{n-1}\})\} \end{array}$$

Thus, to compute the powerset of a set S (represented in TLC$^=$as an iterator), we start out with the list $[Nil] \equiv \lambda c.\, \lambda n.\, c\,(\lambda c'.\, \lambda n'.\, n')\, n$ containing just the empty list, and then perform an iteration over the elements of S, computing at each step the union of the list P produced so far and the list P' derived from P by prepending the current element of S to every element of P:

$$
\begin{aligned}
&\mathit{SimplePowerset}_{\{\alpha\}}\colon ((\alpha \to \{\{\alpha\}\} \to \{\{\alpha\}\}) \to \{\{\alpha\}\} \to \{\{\alpha\}\}) \to \{\{\alpha\}\} \equiv \\
&\quad \lambda s\colon (\alpha \to \{\{\alpha\}\} \to \{\{\alpha\}\}) \to \{\{\alpha\}\} \to \{\{\alpha\}\}. \\
&\qquad s\,(\lambda x\colon \alpha.\, \lambda p\colon \{\{\alpha\}\}.\, \mathit{Union}_{\{\{\alpha\}\}}\, p\, (\mathit{Prepend}\; x\; p))\, [\mathit{Nil}],
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{Prepend}\colon \alpha \to \{\{\alpha\}\} \to \{\{\alpha\}\} \equiv \\
&\quad \lambda x\colon \alpha.\, \lambda p\colon \{\{\alpha\}\}. \\
&\qquad \lambda c\colon \{\alpha\} \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad p\,(\lambda s\colon \{\alpha\}.\, \lambda p'\colon \tau.\, c\,(\mathit{Cons}\; x\; s)\, p')\, n
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{Cons}\colon \alpha \to \{\alpha\} \to \{\alpha\} \equiv \\
&\quad \lambda x\colon \alpha.\, \lambda s\colon \{\alpha\}. \\
&\qquad \lambda c\colon \alpha \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad c\, x\, (s\, c\, n).
\end{aligned}
$$

SimplePowerset is well-typed and computes the powerset function. However, it is not good enough for use with the other algebra operators, because it introduces an "inhomogeneity" in either input or output types. When encoding a relational expression $\phi(S)$ where S is the argument of a powerset operation, S is used as an iterator, and its type must then be modified (by type substitution) to reflect the type of the output powerset. As a consequence, S may become type-incompatible with relational operators, such as *Equal* and *Member*, which inductively recurse on the structure of complex objects. We now discuss this anomaly in further detail.

In a type such as $\{\alpha\} \equiv (\alpha \to \tau \to \tau) \to \tau \to \tau$, where α corresponds to a non-atomic complex object sort, there may be many free occurrences of the type variable τ in α, each occurrence introduced by one of the various tuple and set constructors used in constructing α. In principle, occurrences of τ belonging to different constructors could correspond to different types; in fact, the most general typing of an encoding of a complex object would use a different free type variable for each constructor. However, structure-traversing operators such as *Equal* or *Member* force all these occurrences to correspond to the same type; for example, equality of sets iterates equality tests over elements of the set, equality of tuples iterates equality over tuple components, and so on. An expression $(\mathit{Equal}_{\{\alpha\}}\; S\; S')$ can be typed with any type β substituted for τ, as long as the substitution is carried out for *all* occurrences of τ. Unfortunately, the *SimplePowerset* operator forces the type of its input to be $(\alpha \to \{\{\alpha\}\} \to \{\{\alpha\}\}) \to \{\{\alpha\}\} \to \{\{\alpha\}\}$, i.e., $\{\alpha\}$ with $\{\{\alpha\}\}$ substituted for the *topmost* occurrences of τ only; since *SimplePowerset* never needs to examine the structure of the *elements* of the input set—an instance of parametric polymorphism—it need not mutate the type of the elements, as appearing in the input type or the output type. Therefore, once a set S is used as input to $\mathit{SimplePowerset}_{\{\alpha\}}$, it cannot be used as input to any operator involving *Equal* or *Member*.

We could resolve this problem by assigning the "homogeneous" type $\{\alpha\}[\tau := \{\{\alpha\}\}]$—read "$\{\alpha\}$ with *all* occurrences of τ replaced by $\{\{\alpha\}\}$"—to the input of *SimplePowerset*, but then the type of its output would become $\{\{\alpha[\tau := \{\{\alpha\}\}]\}\}$, i.e., $\{\{\alpha\}\}$ with $\{\{\alpha\}\}$ substituted for all *but the topmost* occurrences of τ, and now the output would be unsuitable for further use with *Equal* or *Member*. In summary, *SimplePowerset* spoils either its input or output—it cannot be freely combined with the other algebra operators.

5.3.1 A Type-Homogeneous Powerset

To restore homogeneity to the types, we construct a *Powerset* operator of type $\{\alpha\}\,[\tau\!:=\{\{\alpha\}\}] \to \{\{\alpha\}\}$, where the type of the input is $\{\alpha\}$ with *all* occurrences of τ replaced by $\{\{\alpha\}\}$ and the type of the output is $\{\{\alpha\}\}$ with *no* substitutions for τ. The technology used to effect this solution is an extension of the "type-laundering" technique introduced in Section 4. It is unavoidable to "raise" the type of the input to *Powerset* in order to exponentiate the input size, but we can at least try to keep both input and output types homogeneous, so that they are usable with other operators. The following variant of *SimplePowerset* achieves this homogeneity:

$$\begin{aligned}
&\mathit{Powerset}_{\{\alpha\}}\colon \{\alpha\}\,[\tau\!:=\{\{\alpha\}\}] \to \{\{\alpha\}\} \equiv\\
&\quad \lambda s\colon \{\alpha\}\,[\tau\!:=\{\{\alpha\}\}].\\
&\qquad s\,(\lambda x\colon \alpha\,[\tau\!:=\{\{\alpha\}\}].\,\lambda p\colon \{\{\alpha\}\}.\,\mathit{Union}_{\{\{\alpha\}\}}\,p\,(\mathit{Prepend}\,(\mathit{Copy}_{\alpha,\{\{\alpha\}\}}\,x)\,p))\,[\mathit{Nil}],
\end{aligned}$$

where *Prepend* is defined as above.

The only modification from *SimplePowerset* is the use of $(\mathit{Copy}_{\alpha,\{\{\alpha\}\}}\,x)$ instead of x as an argument to *Prepend*. The $\mathit{Copy}_{\alpha,\{\{\beta\}\}}$ operator, defined below, computes the identity function on complex objects of sort α, such that its input has type $\alpha\,[\tau\!:=\{\{\beta\}\}]$, whereas its output has type just α. The use of $(\mathit{Copy}_{\alpha,\{\{\alpha\}\}}\,x)$ instead of x does not change the semantics of the *Powerset* operator, while assuring that the output has the "clean" type $\{\{\alpha\}\}$ instead of $\{\{\alpha\,[\tau\!:=\{\{\alpha\}\}]\}\}$.

As in Section 4, we construct $\mathit{Copy}_{\alpha,\{\{\beta\}\}}$ by taking a straightforward "deep copy" operator $\mathit{SimpleCopy}_{\alpha}$ of type $\alpha \to \alpha$ and inserting suitable dummy abstractions and applications to change all subexpressions of type τ to expressions of type $\{\{\beta\}\}$. We first define $\mathit{SimpleCopy}_{\alpha}$ by structural induction over α.

$$\begin{aligned}
&\mathit{SimpleCopy}_{\mathsf{o}}\colon \mathsf{o} \to \mathsf{o} \equiv\\
&\quad \lambda x\colon \mathsf{o}.\, x
\end{aligned}$$

$$\begin{aligned}
&\mathit{SimpleCopy}_{\{\alpha\}}\colon \{\alpha\} \to \{\alpha\} \equiv\\
&\quad \lambda s\colon \{\alpha\}.\\
&\qquad \lambda c\colon \alpha \to \tau \to \tau.\,\lambda n\colon \tau.\\
&\qquad\quad s\,(\lambda x\colon \alpha.\,\lambda p\colon \tau.\,c\,(\mathit{SimpleCopy}_{\alpha}\,x)\,p)\,n
\end{aligned}$$

$$\begin{aligned}
&\mathit{SimpleCopy}_{\alpha_1\times\cdots\times\alpha_k}\colon \alpha_1 \times \cdots \times \alpha_k \to \alpha_1 \times \cdots \times \alpha_k \equiv\\
&\quad \lambda t\colon \alpha_1 \times \cdots \times \alpha_k.\\
&\qquad \lambda f\colon \alpha_1 \to \cdots \to \alpha_k \to \tau.\\
&\qquad\quad t\,(\lambda x_1\colon \alpha_1 \ldots \lambda x_k\colon \alpha_k.\,f\,(\mathit{SimpleCopy}_{\alpha_1}\,x_1)\ldots(\mathit{SimpleCopy}_{\alpha_k}\,x_k))
\end{aligned}$$

It is straightforward to verify that this operator computes the identity on complex objects. If we now want to "raise" the input type by substituting $\{\{\beta\}\}$ for τ, we have to prepend dummy λ-abstractions to the value computed at each level of the copy operation and to get rid of them again at the next higher level by supplying dummy arguments. Thus, the correct *Copy* operator looks like this:

$$\begin{aligned}
&\mathit{Copy}_{\mathsf{o},\{\{\beta\}\}}\colon \mathsf{o}\,[\tau\!:=\{\{\beta\}\}] \to \mathsf{o} \equiv\\
&\quad \lambda x\colon \mathsf{o}.\, x
\end{aligned}$$

$$
\begin{aligned}
& \mathit{Copy}_{\{\alpha\},\{\{\beta\}\}} \colon \{\alpha\}\,[\tau := \{\{\beta\}\}] \rightarrow \{\alpha\} \equiv \\
& \quad \lambda s \colon \{\alpha\}\,[\tau := \{\{\beta\}\}]. \\
& \qquad \lambda c \colon \alpha \rightarrow \tau \rightarrow \tau.\, \lambda n \colon \tau. \\
& \qquad\quad s\ \ (\lambda x \colon \alpha\,[\tau := \{\{\beta\}\}].\, \lambda p \colon \{\{\beta\}\}. \\
& \qquad\qquad\quad \lambda c' \colon \{\beta\} \rightarrow \tau \rightarrow \tau.\, \lambda n' \colon \tau. \\
& \qquad\qquad\qquad c\,(\mathit{Copy}_{\alpha,\{\{\beta\}\}}\, x)\,(p\, c'\, n')) \\
& \qquad\qquad (\lambda c' \colon \{\beta\} \rightarrow \tau \rightarrow \tau.\, \lambda n' \colon \tau.\, n) \\
& \qquad\qquad (\lambda s' \colon \{\beta\}.\, \lambda p' \colon \tau.\, p')\, n
\end{aligned}
$$

To understand how $\mathit{Copy}_{\{\alpha\},\{\{\beta\}\}}$ works, note that if $s \equiv \lambda c.\, \lambda n.\, c\, x_1\, (c\, x_2\, (\cdots (c\, x_k\, n) \cdots))$, then the term $s\,(\lambda x.\, \lambda p.\, \lambda c'.\, \lambda n'.\, c\,(\mathit{Copy}\, x)\,(p\, c'\, n'))$ reduces to $\lambda n.\, \lambda c'.\, \lambda n'.\, c\, y_1\, (c\, y_2\, (\cdots (c\, y_k\, (n\, c'\, n')) \cdots))$, where y_i is a copy of x_i. Applying this term to $\lambda c'.\, \lambda n'.\, n$ yields $\lambda c'.\, \lambda n'.\, c\, y_1\, (c\, y_2\, (\cdots (c\, y_k\, n) \cdots))$, a term where the bound variables c' and n' do not occur free in the body; they are removed by application to dummy arguments of the correct type.

The *Copy* operator for products is defined in a similar spirit:

$$
\begin{aligned}
& \mathit{Copy}_{\alpha_1 \times \cdots \times \alpha_k,\{\{\beta\}\}} \colon (\alpha_1 \times \cdots \times \alpha_k)\,[\tau := \{\{\beta\}\}] \rightarrow \alpha_1 \times \cdots \times \alpha_k \equiv \\
& \quad \lambda t \colon (\alpha_1 \times \cdots \times \alpha_k)\,[\tau := \{\{\beta\}\}]. \\
& \qquad \lambda f \colon \alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow \tau. \\
& \qquad\quad t\ \ (\lambda x_1 \colon \alpha_1\,[\tau := \{\{\beta\}\}] \ldots \lambda x_k \colon \alpha_k\,[\tau := \{\{\beta\}\}]. \\
& \qquad\qquad\quad \lambda c' \colon \{\beta\} \rightarrow \tau \rightarrow \tau.\, \lambda n' \colon \tau. \\
& \qquad\qquad\qquad f\,(\mathit{Copy}_{\alpha_1,\{\{\beta\}\}}\, x_1) \ldots (\mathit{Copy}_{\alpha_k,\{\{\beta\}\}}\, x_k)) \\
& \qquad\qquad (\lambda s' \colon \{\beta\}.\, \lambda p' \colon \tau.\, p')\, n \qquad\qquad \text{(where } n \text{ is some variable of type } \tau\text{)}
\end{aligned}
$$

An induction on the structure of encodings of complex objects shows that the above terms are well-typed and that they compute the identity function on encodings of complex objects of sort α, such that the input encoding can be typed as $\alpha\,[\tau := \{\{\beta\}\}]$ and the output encoding can be typed as α. It follows that the $\mathit{Powerset}_{\{\alpha\}}$ operator given above can be typed as shown and that it indeed computes the powerset function on encodings of complex objects of sort $\{\alpha\}$.

5.3.2 Powersets, Sharing, and Type Compatibility

There is, however, a further technical issue requiring attention: when a relational expression involving *Powerset* is *shared* in a larger computation, the respective contexts in which the expression occurs may introduce conflicting type constraints. While the input and output types of *Powerset* are now "homogeneous" in the sense that all occurrences of τ correspond to the same type, an expression $(\mathit{Powerset}_{\{\alpha\}}\, S)$ still forces the substitution $\tau := \{\{\alpha\}\}$ on the type of S. This substitution constrains the type of objects that S can output when S is used as an iterator in another *Powerset* operator, for example $(\mathit{Powerset}_{\{\alpha \times \alpha\}}\, (\mathit{Times}\, S\, S))$.

Other combinations of S with the output of $(\mathit{Powerset}_{\{\alpha\}}\, S)$ become equally problematic. For example, the expression $(\mathit{Equal}\, (\mathit{Set}\, S)\, (\mathit{Powerset}_{\{\alpha\}}\, S))$ would not type because $(\mathit{Powerset}_{\{\alpha\}}\, S)$ has type $\{\{\alpha\}\}$, whereas $(\mathit{Set}\, S)$ would be typed as $\{\{\alpha\}\}\,[\tau := \{\{\alpha\}\}]$ due to the "contaminated" type of S. To resolve this problem, we can insert a "type-laundering" operator whenever a conflict occurs. For example, the expression $(\mathit{Equal}\, (\mathit{Set}\, S)\, (\mathit{Powerset}_{\{\alpha\}}\, S))$ can be typed if it is rewritten as $(\mathit{Equal}\, (\mathit{Set}\, (\mathit{Copy}_{\{\alpha\},\{\{\alpha\}\}}\, S)\, (\mathit{Powerset}_{\{\alpha\}}\, S))$.

To make this "type-laundering" process precise, let us fix an expression $\phi(r_1, \ldots, r_k)$ in the complex object algebra, and consider its compilation into a TLC$^=$ term. We may think of ϕ as a circuit, where the relational variables r_j are inputs, and the algebra operators are gates. Observe that the only shared subexpressions are the inputs. Since the circuit is acyclic, we can topologically enumerate the inputs and gates, such that each gate is given an index greater than those of its inputs. For simplicity, we further stipulate that the input relations are given minimal indices; let j be the index of r_j, $1 \le j \le k$. Assume that there are n internal gates and inputs, labeled $1, 2, \ldots, n$, and among these m *Powerset* gates, with indices $k < \nu_1 < \cdots < \nu_m \le n$. Write $[i, j]$ for the interval $i, i+1, \ldots, j$ and $(i, j]$ for $[i+1, j]$.

In order to construct a typing for a TLC$^=$ term realizing ϕ, we divide the gates in the circuit into $m+1$ *layers* of circuitry delineated by the placement of *Powerset* gates in the topological ordering: we define $\ell_1 = [1, \nu_1]$, $\ell_2 = (\nu_1, \nu_2]$, $\ell_3 = (\nu_2, \nu_3]$, $\ldots$, $\ell_m = (\nu_{m-1}, \nu_m]$, $\ell_{m+1} = (\nu_m, n]$. The *boundaries* between successive layers can be thought of graphically, where a single *Powerset* sits on the boundary, and wires between gates from different layers cross the boundaries. Each boundary naturally defines *inputs* (equivalently, *outputs*) between adjacent layers, where the *Powerset* gate labeled $\nu_j \in \ell_j$ generates an input to layer ℓ_{j+1}, and the remaining inputs are the wires crossing that boundary. Each wire can be labeled with a complex algebra sort that is compatible with the gates at its endpoints. Note that a wire crosses several boundaries if the gates at its endpoints are in nonadjacent layers.

Lemma 5.9 *Given an r-layer circuit, there exists a TLC$^=$ term simulating the circuit where for each input of sort σ in the circuit, the TLC$^=$ term has a corresponding free variable of type $\sigma[\tau := \{\{\alpha_1\}\}][\tau := \{\{\alpha_2\}\}]\cdots[\tau := \{\{\alpha_r\}\}]$, such that each substitution $[\tau := \{\{\alpha_j\}\}]$ is generated from the jth* Powerset *operator.*

Proof: We construct a TLC$^=$ term E from the circuit in a syntax-directed manner, via an inductive construction on the number of layers.

The inductive construction has two cases for the basis $r = 1$. If no *Powerset* gate appears in the circuit, the construction is immediate and trivial. If one *Powerset* gate appears, it must be the gate generating the output of the circuit, and of sort $\{\{\alpha_1\}\}$. We construct the TLC$^=$ term E identical to the circuit, where each wire of sort σ in the circuit *except the output wire* is realized by a subterm with type $\sigma[\tau := \{\{\alpha_1\}\}]$ (via type substitution), and the entire term has type $\{\{\alpha_1\}\}$. Preserving the invariant, the input relation r_j of sort ρ_j appears in E as a free variable of type $\rho_j[\tau := \{\{\alpha_1\}\}]$ if there is a *Powerset* gate, and simply of type ρ_j otherwise.

When there is more than one layer in the circuit, we divide it into the "top" part comprising layers $\ell_r, \ell_{r-1}, \ldots, \ell_2$, and the "bottom" part consisting of layer ℓ_1. By induction, the top part is realized by a TLC$^=$ term E', where for each input wire w to the top part (i.e., to ℓ_2) having sort σ_w, the term E' has a corresponding free variable x_w of type $\sigma_w[\tau := \{\{\alpha_2\}\}]\cdots[\tau := \{\{\alpha_r\}\}]$. We now need to use the structure of the bottom part to generate TLC$^=$ terms that are type-compatible with the x_w.

Similar to the basis construction with one *Powerset* gate, *every* wire u of sort σ_u from the bottom layer ℓ_1, *except the output wire P of the Powerset gate with sort* $\{\{\alpha_1\}\}$, can be realized by a TLC$^=$ term T_u of type $\sigma_u[\tau := \{\{\alpha_1\}\}]$. The output of the *Powerset* gate is realized by a term T_P of type $\{\{\alpha_1\}\}$, and every other output wire w from the bottom layer can be realized by the TLC$^=$ term $(\mathit{Copy}_{\sigma_w, \{\{\alpha_1\}\}}\, T_w)$ of type σ_w.

Under the type substitution $\Sigma = [\tau := \{\{\alpha_2\}\}]\cdots[\tau := \{\{\alpha_r\}\}]$, T_P can then be given type $\{\{\alpha_1\}\}[\tau := \{\{\alpha_2\}\}]\cdots[\tau := \{\{\alpha_r\}\}]$ to be compatible with its corresponding input wire x_P in the top layer, and $(\mathit{Copy}_{\sigma_w, \{\{\alpha_1\}\}}\, T_w)$ can be given type $\sigma_w[\tau := \{\{\alpha_2\}\}]\cdots[\tau := \{\{\alpha_r\}\}]$ that is type-compatible with input wire x_w. Furthermore, every input to the bottom layer of sort σ_u and type

$\sigma_u\,[\tau{:=}\,\{\{\alpha_1\}\}]$ can then be given type $\sigma_u\,[\tau{:=}\,\{\{\alpha_1\}\}]\,[\tau{:=}\,\{\{\alpha_2\}\}]\cdots[\tau{:=}\,\{\{\alpha_r\}\}]$, preserving the inductive hypothesis. □

Example 5.10 Suppose we want to use this compilation process to translate the complex object algebra expression $(\mathit{Set}\,S) \cap (\mathit{Powerset}\,S)$, where S is of sort $\{\alpha\}$, into TLC$^=$. Let use enumerate the operators in this expression topologically as: S, *Set*, *Powerset*, $\cap$. The corresponding circuit has two layers, separated by a *Powerset* gate: the top layer contains an intersection gate and the bottom layer contains an input node S and a *Set* gate. The inputs to the top layer are the output of the *Powerset* gate and a wire from the *Set* gate.

We realize the top layer by the term $(\mathit{Intersection}_{\{\{\alpha\}\}}\,x_1\,x_2)$ and its inputs by the terms $T_1 \equiv (\mathit{Set}_{\{\alpha\}}\,S)$ and $T_2 \equiv (\mathit{Powerset}_{\{\alpha\}}\,S)$, where S is typed as $\{\alpha\}\,[\tau{:=}\,\{\{\alpha\}\}]$. Term T_1 is of type $\{\{\alpha\}\}\,[\tau{:=}\,\{\{\alpha\}\}]$ and term T_2 is of type $\{\{\alpha\}\}$. When forming the combined expression according to the rules above, T_2 is substituted directly for x_2 and a copy of T_1 is substituted for x_1. Thus, the TLC$^=$ representation of the circuit becomes

$$(\mathit{Intersection}_{\{\{\alpha\}\}}\,(\mathit{Copy}_{\{\{\alpha\}\},\{\{\alpha\}\}}\,(\mathit{Set}_{\{\alpha\}}\,S))\,(\mathit{Powerset}_{\{\alpha\}}\,S)).$$

Note that a different, but functionally equivalent, term is generated from the topological ordering S, *Powerset*, *Set*, $\cap$.

5.3.3 Removing Polymorphic Equality

The procedure given above produces a typable embedding of any complex object algebra expression into TLC$^=$, but it has a minor aesthetic defect: it requires a polymorphic typing of the *Eq* predicate. This is because an occurrence of *Eq* in an operator at level l must be typed as $\mathsf{o} \to \mathsf{o} \to \gamma_l \to \gamma_l \to \gamma_l$ instead of $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$, where $\gamma_l = \tau\,[\tau{:=}\,\{\{\alpha_l\}\}]\,[\tau{:=}\,\{\{\alpha_{l+1}\}\}]\cdots[\tau{:=}\,\{\{\alpha_m\}\}]$. If we want the type of *Eq* to stay monomorphic (for example, if we want to supply *Eq* as part of the input instead of relying on a built-in predicate, cf. Section 2.4), we can do this by replacing every occurrence of *Eq* at level l with the term

$$\begin{aligned}
&Eq_l{:}\,\mathsf{o} \to \mathsf{o} \to \gamma_l \to \gamma_l \to \gamma_l \equiv\\
&\quad \lambda x{:}\,\mathsf{o}.\,\lambda y{:}\,\mathsf{o}.\\
&\qquad \lambda u{:}\,\gamma_l.\,\lambda v{:}\,\gamma_l.\\
&\qquad\quad \lambda z_1{:}\,\beta_1 \ldots \lambda z_k{:}\,\beta_k.\, Eq\,x\,y\,(u\,z_1 \ldots z_k)\,(v\,z_1 \ldots z_k),
\end{aligned}$$

where the types $\beta_1, \ldots, \beta_k$ are given by $\gamma_l = \beta_1 \to \cdots \to \beta_k \to \tau$. For every pair of constants o_i, o_j, the normal form of $(Eq\,o_i\,o_j)$ arises from the normal form of $(Eq_l\,o_i\,o_j)$ by a series of η-reductions [4]. It is known that in a $\beta\eta$-reduction sequence, η-reductions can always be pushed to the end [4, Theorem 15.1.6]. This remains true even if *Eq*-reductions are added, because *Eq*- and η-reductions commute for well-typed terms. Thus, the normal form of a query using Eq_l instead of *Eq* η-reduces to the normal form of the original query. However, the output of a query is always the encoding $\overline{R}$ of a complex object having an "uncontaminated" type, and it is easy to see that there is no well-typed term other than $\overline{R}$ that η-reduces to $\overline{R}$. Thus, it follows that the normal form of a query using Eq_l instead of *Eq* and the normal form of the original query are identical.

In summary, we have shown:

Theorem 5.11 *Any ALG query q can be embedded into TLC$^=$. If the Powerset operator is not used in q, the embedding is PTIME in the sense of Definition 2.3.*

6 Embeddings in Low-Order Fragments of $TLC^=$

In this section, we modify the simulation of fixpoint queries in Section 4 to minimize the functionality order of query terms. We first revise our input-output conventions and we then revisit our embeddings of relational algebra and PTIME queries.

Minimizing the input/output order: In the encoding scheme used so far, a k-ary relation is encoded as a term

$$\begin{array}{l} \lambda c{:}\,((\mathsf{o} \to \cdots \to \mathsf{o} \to \tau) \to \tau) \to \tau \to \tau.\, \lambda n{:}\,\tau. \\ \quad (c\,(\lambda f{:}\,\mathsf{o} \to \cdots \to \mathsf{o} \to \tau.\, f\, o_{1,1}\, o_{1,2} \ldots o_{1,k}) \\ \qquad (c\,(\lambda f{:}\,\mathsf{o} \to \cdots \to \mathsf{o} \to \tau.\, f\, o_{2,1}\, o_{2,2} \ldots o_{2,k}) \\ \qquad\quad \ldots \\ \qquad\qquad (c\,(\lambda f{:}\,\mathsf{o} \to \cdots \to \mathsf{o} \to \tau.\, f\, o_{m,1}\, o_{m,2} \ldots o_{m,k})\, n) \ldots)) \end{array}$$

whose type $\{\mathsf{o}^k\}$ is of order 4. The relational operators of Section 3 are coded as terms of type $\{\mathsf{o}^k\} \to \{\mathsf{o}^l\}$ or $\{\mathsf{o}^k\} \to \{\mathsf{o}^l\} \to \{\mathsf{o}^m\}$, which are of order 5. The fixpoint queries of Section 4 are of even higher order because of type contamination—their type is $(\{\mathsf{o}^{k_1}\} \to \cdots \to \{\mathsf{o}^{k_l}\})\,[\tau := \{\mathsf{o}^k\}] \to \{\mathsf{o}^k\}$, which is of order 8. Finally, the order of complex object queries is unbounded, because the type structure essentially reflects the nesting of tuples and sets in the input and output.

Although these encodings are quite natural, they are somewhat wasteful with respect to the order of their types. A λ-term of order 4 is already sufficiently powerful to exponentiate a Church numeral, so the use of an order 8 term to compute a polynomial fixpoint seems unnecessary. Indeed, by changing the input-output conventions slightly, we can do much better.

The revised encoding scheme does away with independent representations for tuples and lists. The only data structure is the *relation*, which is encoded as follows. Let $\mathcal{D} = \{o_1, o_2, \ldots\}$ be the set of constants. For convenience, we assume that this set of constants also serves as the universe over which relations are defined. We replace Definition 2.1 with the following:

Definition 6.1 *Let $R = \{(o_{1,1}, o_{1,2}, \ldots, o_{1,k}), (o_{2,1}, o_{2,2}, \ldots, o_{2,k}), \ldots, (o_{m,1}, o_{m,2}, \ldots, o_{m,k})\} \subseteq \mathcal{D}^k$ be a k-ary relation over $\mathcal{D}$. An* encoding *of R, denoted by $\overline{R}$, is a λ-term:*

$$\begin{array}{l} \lambda c{:}\,\overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \tau \to \tau.\, \lambda n{:}\,\tau. \\ \quad (c\, o_{1,1}\, o_{1,2} \ldots o_{1,k} \\ \qquad (c\, o_{2,1}\, o_{2,2} \ldots o_{2,k} \\ \qquad\quad \ldots \\ \qquad\qquad (c\, o_{m,1}\, o_{m,2} \ldots o_{m,k}\, n) \ldots)) \end{array}$$

in which every tuple of R appears exactly once. Query terms *are defined as in Definition 2.1.*

The difference between this encoding scheme and the one of Definition 2.1 is that the fields of a tuple now appear as separate atomic arguments to the "list-walking" function c, instead of being hidden in a single functional argument. One can think of $\overline{R}$ as a generalized Church numeral that not only iterates a given function a certain number of times, but also provides different data at each iteration.

If R contains at least two tuples, the principal type of $\overline{R}$ is $(\mathsf{o} \to \cdots \to \mathsf{o} \to \rho \to \rho) \to \rho \to \rho$, where ρ is a type variable.[3] The order of this type is 2, independent of the arity of R. We abbreviate this type as o_k^ρ. Instances of this type, obtained by substituting some type expression α for ρ, are abbreviated as o_k^α, or, if the exact nature of α does not matter, as o_k^*.

Theorem 6.2 *Any relational algebra query can be PTIME-embedded into order 3 TLC$^=$ in the sense of Definition 2.3 (but with encoding scheme 6.1 instead of 2.1).*

Proof: The encodings of the relational operators can be adopted to our new input-output convention in a straightforward fashion. The same also applies for the reduction strategies from Section 3. The basic idea is that whenever an operator was passed a k-tuple as an argument, it is now passed the k individual components of the tuple instead. The revised encodings are as follows:

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

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

$$
\begin{aligned}
&\mathit{Intersection}_k{:}\,\mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \equiv \\
&\quad \lambda r{:}\mathsf{o}_k^\tau.\, \lambda r'{:}\mathsf{o}_k^\tau. \\
&\qquad \lambda c{:}\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, \lambda n{:}\,\tau. \\
&\qquad\quad r\, (\lambda x_1{:}\mathsf{o} \ldots \lambda x_k{:}\mathsf{o}.\, \lambda p{:}\,\tau.\, (\mathit{Member}\, x_1 \ldots x_k\, r')\, (c\, x_1 \ldots x_k\, p)\, p)\, n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Setminus}_k{:}\,\mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \equiv \\
&\quad \lambda r{:}\mathsf{o}_k^\tau.\, \lambda r'{:}\mathsf{o}_k^\tau. \\
&\qquad \lambda c{:}\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, \lambda n{:}\,\tau. \\
&\qquad\quad r\, (\lambda x_1{:}\mathsf{o} \ldots \lambda x_k{:}\mathsf{o}.\, \lambda p{:}\,\tau.\, (\mathit{Member}\, x_1 \ldots x_k\, r')\, p\, (c\, x_1 \ldots x_k\, p))\, n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Union}_k{:}\,\mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \equiv \\
&\quad \lambda r{:}\mathsf{o}_k^\tau.\, \lambda r'{:}\mathsf{o}_k^\tau. \\
&\qquad \lambda c{:}\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, \lambda n{:}\,\tau. \\
&\qquad\quad r\, (\lambda x_1{:}\mathsf{o} \ldots \lambda x_k{:}\mathsf{o}.\, \lambda p{:}\,\tau.\, (\mathit{Member}_k\, x_1 \ldots x_k\, r')\, p\, (c\, x_1 \ldots x_k\, p))\, (r'\, c\, n)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{Times}_{k,l} \colon \mathsf{o}_k^\tau \to \mathsf{o}_l^\tau \to \mathsf{o}_{k+l}^\tau \equiv \\
&\quad \lambda r \colon \mathsf{o}_k^\tau .\, \lambda r' \colon \mathsf{o}_l^\tau . \\
&\qquad \lambda c \colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau .\, \lambda n \colon \tau . \\
&\qquad\quad r\,(\lambda x_1 \colon \mathsf{o} \ldots \lambda x_k \colon \mathsf{o} .\, \lambda p \colon \tau . \\
&\qquad\qquad r'\,(\lambda y_1 \colon \mathsf{o} \ldots \lambda y_l \colon \mathsf{o} .\, \lambda p' \colon \tau . \\
&\qquad\qquad\qquad c\, x_1 \ldots x_k\, y_1 \ldots y_l\, p')\, p)\, n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Select}_{k;i=j} \colon \mathsf{o}_k^\tau \to \mathsf{o}_k^\tau \equiv \\
&\quad \lambda r \colon \mathsf{o}_k^\tau . \\
&\qquad \lambda c \colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau .\, \lambda n \colon \tau . \\
&\qquad\quad r\,(\lambda x_1 \colon \mathsf{o} \ldots \lambda x_k \colon \mathsf{o} .\, \lambda p \colon \tau .\, (\mathit{Eq}\, x_i\, x_j)\,(c\, x_1 \ldots x_k\, p)\, p)\, n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Project}_{k;i_1,\ldots,i_l} \colon \mathsf{o}_k^\tau \to \mathsf{o}_l^\tau \equiv \\
&\quad \lambda r \colon \mathsf{o}_k^\tau . \\
&\qquad \lambda c \colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau .\, \lambda n \colon \tau . \\
&\qquad\quad r\,(\lambda x_1 \colon \mathsf{o} \ldots \lambda x_k \colon \mathsf{o} .\, \lambda p \colon \tau . \\
&\qquad\qquad r\,(\lambda y_1 \colon \mathsf{o} \ldots \lambda y_k \colon \mathsf{o} .\, \lambda p' \colon \tau . \\
&\qquad\qquad\qquad (\mathit{Equal}_l\, x_{i_1} \ldots x_{i_l}\, y_{i_1} \ldots y_{i_l})\ \ (\mathit{Equal}_k\, x_1 \ldots x_k\, y_1 \ldots y_k\, (c\, x_{i_1} \ldots x_{i_l}\, p)\, p)\ \ p')\, p)\, n
\end{aligned}
$$

With these terms, we can encode any relational algebra query in TLC$^=$ of order 3. □

Representing intermediate results more efficiently: Under the revised input-output format, the order of a fixpoint query is

$$
\mathrm{order}\,(\mathsf{o}_{k_1}^\tau\,[\tau \colon= \mathsf{o}_k^\tau] \to \cdots \to \mathsf{o}_{k_l}^\tau\,[\tau \colon= \mathsf{o}_k^\tau] \to \mathsf{o}_k^\tau) = 5.
$$

With the following trick, we can bring the order down to 4: During the fixpoint computation, we represent intermediate values of the output relation not as iterators, but as *characteristic functions*. A characteristic function representation of a k-ary relation R is a TLC$^=$ term

$$
C_R \colon \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \mathsf{Bool}
$$

such that for any k constants $o_{i_1}, \ldots, o_{i_k}$,

$$
(C_R\, o_{i_1} \ldots o_{i_k}) \rhd \begin{cases} \mathit{True} & \text{if } (o_{i_1}, \ldots, o_{i_k}) \in R, \\ \mathit{False} & \text{if } (o_{i_1}, \ldots, o_{i_k}) \notin R. \end{cases}
$$

Note that the type of C_R, $\mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{Bool}$, is of order 1. We abbreviate this type as χ_k.

List representations of relations can be transformed into characteristic function representations by the following family of operators (one for each arity):

$$
\begin{aligned}
&\mathit{ListToChar}_k \colon \mathsf{o}_k^\tau \to \chi_k \equiv \\
&\quad \lambda r \colon \mathsf{o}_k^\tau . \\
&\qquad \lambda x_1 \colon \mathsf{o} \ldots \lambda x_k \colon \mathsf{o} .\, (\mathit{Member}_k\, x_1 \ldots x_k\, r)
\end{aligned}
$$

In order to translate back from a characteristic function to an iterator, we need a list $A{:}\,\mathsf{o}_1^\tau$ containing all the constants in the active domain. (Such a list can be obtained from the input relations by forming the union of all columns, using the *Project* and *Union* operators.) Given this list, the iterator corresponding to a characteristic function $C_R{:}\,\chi_k$ is computed by the TLC$^=$ term

$$
\begin{array}{l}
\mathit{CharToList}_k{:}\,\chi_k \rightarrow \mathsf{o}_k^\tau \equiv \\
\quad \lambda f{:}\,\mathsf{o} \rightarrow \cdots \rightarrow \mathsf{o} \rightarrow \mathsf{Bool}. \\
\qquad \lambda c{:}\,\mathsf{o} \rightarrow \cdots \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau.\, \lambda n{:}\,\tau. \\
\qquad\quad A\,(\lambda x_1{:}\,\mathsf{o}.\, \lambda p_1{:}\,\tau. \\
\qquad\qquad A\,(\lambda x_2{:}\,\mathsf{o}.\, \lambda p_2{:}\,\tau. \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad A\,(\lambda x_k{:}\,\mathsf{o}.\, \lambda p_k{:}\,\tau. \\
\qquad\qquad\qquad\quad f\,x_1 \ldots x_k\,(c\,x_1 \ldots x_k\,p_k)\,p_k)\,p_{k-1}) \ldots p_1)\,n.
\end{array}
$$

Theorem 6.3 *Any PTIME database query can be encoded in order 4 TLC$^=$.*

Proof: We proceed by modifying the proof of Section 4 in view of our PTIME-embedding of relational algebra in TLC$^=$ or order 3.

Suppose that the TLC$^=$ term $(\lambda r.\, Q){:}\,\mathsf{o}_k^\tau \rightarrow \mathsf{o}_k^\tau$ of order 3 represents a relational query to be iterated, with variables $r_1, \ldots, r_l$ representing the input relations occurring free in Q. The term $\lambda f.\, Q'$, where

$$Q' \equiv \mathit{ListToChar}_k\,(Q\,[r{:=}\,(\mathit{CharToList}_k\,f)]),$$

represents the same query, but its order is only 2!

To express the fixpoint of $\lambda f.\, Q'$, we would like to define, similar to Section 4,

$$\mathit{Fix}_Q \equiv \lambda r_1 \ldots \lambda r_l.\, \mathit{CharToList}_k\,(\mathit{Crank}\,(\lambda f.\, Q')\,(\mathit{ListToChar}_k\,\mathit{Nil})),$$

where *Crank* is a suitably large Church numeral derived from a Cartesian product of the active domain and *Nil* is the empty list. However, this term can be typed only with a certain amount of "type-laundering." More precisely, the occurrences of r_i in *Crank* have to have a "contaminated" type $(\mathsf{o}_{k_i}^\tau)[\tau{:=}\,\chi_k]$, whereas the occurrences of r_i in Q' and *CharToList* have to have a "clean" type $\mathsf{o}_{k_i}^\tau$, for $1 \leq i \leq l$. This can be achieved by replacing the latter occurrences of r_i by the term $(\mathit{Copy}_{k_i}\,r_i)$, where Copy_{k_i} is the following "type-laundering" operator:

$$
\begin{array}{l}
\mathit{Copy}_{k_i}{:}\,(\mathsf{o}_{k_i}^\tau)\,[\tau{:=}\,\chi_k] \rightarrow \mathsf{o}_{k_i}^\tau \equiv \\
\quad \lambda r{:}\,(\mathsf{o}_{k_i}^\tau)\,[\tau{:=}\,\chi_k]. \\
\qquad \lambda c{:}\,\mathsf{o} \rightarrow \cdots \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau.\, \lambda n{:}\,\tau. \\
\qquad\quad (r\ (\lambda x_1{:}\,\mathsf{o} \ldots \lambda x_{k_i}{:}\,\mathsf{o}.\, \lambda p{:}\,\chi_k. \\
\qquad\qquad\qquad \lambda y_1{:}\,\mathsf{o} \ldots \lambda y_k{:}\,\mathsf{o}.\, \lambda u{:}\,\tau.\, \lambda v{:}\,\tau.\, c\,x_1 \ldots x_{k_i}\,(p\,y_1 \ldots y_k\,u\,v)) \\
\qquad\qquad (\lambda y_1{:}\,\mathsf{o} \ldots \lambda y_k{:}\,\mathsf{o}.\, \lambda u{:}\,\tau.\, \lambda v{:}\,\tau.\, n))\,z_1 \ldots z_k\,n\,n
\end{array}
$$

(Here, $z_1, \ldots, z_k$ are some variables of type o.) The proof of correctness is similar to that of Section 4.2.

Thus, the final encoding of the fixpoint query becomes

$$
\begin{aligned}
&\mathit{Fix}_Q\colon (\mathsf{o}^{\tau}_{k_1} \to \cdots \to \mathsf{o}^{\tau}_{k_l})\,[\tau\colon= \chi_k] \to \mathsf{o}^{\tau}_k \equiv \\
&\quad \lambda r_1\colon \mathsf{o}^{\chi_k}_{k_1} \ldots \lambda r_l\colon \mathsf{o}^{\chi_k}_{k_l}. \\
&\qquad \mathit{CharToList}_k\,[r_1\colon= (\mathit{Copy}_{k_1}\, r_1), \ldots, r_l\colon= (\mathit{Copy}_{k_l}\, r_l)] \\
&\qquad\quad (\mathit{Crank} \\
&\qquad\qquad (\lambda f\colon \chi_k.\, Q'\,[r_1\colon= (\mathit{Copy}_{k_1}\, r_1), \ldots, r_l\colon= (\mathit{Copy}_{k_l}\, r_l)]) \\
&\qquad\qquad (\mathit{ListToChar}_k\, \mathit{Nil})).
\end{aligned}
$$

It is easy to see that this term is indeed well-typed and that its type is of order 4. It follows that every inflationary fixpoint query can be encoded in TLC$^=$ of order 4. Observe that the *Precedes* operator of Section 4.2 works without change in our current setting. Thus, Theorem 6.3 follows from the Immerman-Vardi characterization of QPTIME. □

Remark 6.4 Note that Theorem 6.3 does not make any claims about the number of reduction steps needed to normalize the query expression. In fact, it is not obvious at all that a fixpoint query in this new encoding scheme can be evaluated in PTIME. This is due to the fact that for any iterator R representing a relation, the term $(\mathit{ListToChar}\, R)$ reduces to a normal form whose size is exponential in the size of R! Thus, a polynomial evaluation strategy must somehow recognize such terms and use special data structures to store their normal forms efficiently. This program is carried out in [23, 24], where it is shown that for computations over finite structures, TLC$^=$ of order 4 expresses only PTIME computations. Thus, over finite structures, TLC$^=$ of order 4 captures exactly PTIME.

Finally, we observe that by eliminating constants and equality using the technique of Section 2.4, we obtain an embedding of PTIME in the TLC of order 5:

Theorem 6.5 *Let $(\mu_R\,\phi)$ be any fixpoint query mapping relational variables $r_1, \ldots, r_l$ of arities $k_1, \ldots, k_l$ to a relation of arity k. Then there exists a TLC term Q such that whenever $R_1, \ldots, R_l$ are relations of arities $k_1, \ldots, k_l$, the expression $(Q\, \overline{\overline{\mathrm{Eq}}}\, \overline{\overline{R_1}} \ldots \overline{\overline{R_l}})$ can be typed in the order-5 fragment of TLC and it reduces to $\overline{\overline{(\mu_R\,\phi)\,(R_1, \ldots, R_l)}}$. Here, $\overline{\overline{R}}$ denotes the encoding of relation R in the format described above, except that TLC$^=$ constants are replaced by the appropriate projection functions, and $\overline{\overline{\mathrm{Eq}}}$ is a domain-specific encoding of the equality predicate as described in Section 2.4.*

Proof: By Theorem 6.3, there exists a TLC$^=$ query term Fix_ϕ of order 4 encoding the query $(\mu_R\,\phi)$. Let Q be the TLC-term $\lambda \mathit{Eq}\colon \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau.\, \mathit{Fix}_\phi$. Let $\overline{\overline{R_1}}, \ldots, \overline{\overline{R_l}}$ be encodings of relations of arities $k_1, \ldots, k_l$ using projection functions instead of constants, and let $\overline{\overline{\mathit{Eq}}}$ be an encoding of Eq appropriate for these projection functions, as described in Section 2.4. The types assigned to $\overline{\overline{\mathit{Eq}}}$ and $\overline{\overline{R_1}}, \ldots, \overline{\overline{R_l}}$ in the TLC expression $(Q\, \overline{\overline{\mathit{Eq}}}\, \overline{\overline{R_1}} \ldots \overline{\overline{R_l}})$ are exactly the same as the types assigned to Eq and $\overline{R_1}, \ldots, \overline{R_l}$ in the TLC$^=$ expression $(\mathit{Fix}_\phi\, \overline{R_1} \ldots \overline{R_l})$, except that the type constant o is replaced by an order-1 type $\omega \equiv \tau \to \cdots \to \tau$, where the number of τ's depends on the size of the domain. According to the description of fixpoint queries above, the types assigned to $\overline{R_1}, \ldots, \overline{R_l}$ are order-3 types of the form $(\mathsf{o} \to \cdots \to \mathsf{o} \to \chi_k \to \chi_k) \to \chi_k \to \chi_k$, where χ_k is the order-1 type $\mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{Bool}$. Substituting ω for o in this type increases the order by 1, so the occurrences of $\overline{\overline{R_1}}, \ldots, \overline{\overline{R_l}}$ in $(Q\, \overline{\overline{\mathit{Eq}}}\, \overline{\overline{R_1}} \ldots \overline{\overline{R_l}})$ are assigned order-4 types. The type of $\overline{\overline{\mathit{Eq}}}$ is $\omega \to \omega \to \mathsf{Bool}$, which is of order 2. It follows that the occurrence of Q in $(Q\, \overline{\overline{\mathit{Eq}}}\, \overline{\overline{R_1}} \ldots \overline{\overline{R_l}})$, and therefore the entire expression, can be typed using order at most 5. □

Remark 6.6 It is easy to see that the same technique can be applied to query terms in order k $\mathrm{TLC}^=$ for arbitrary $k \geq 3$, leading to re-encodings of such query terms in the "pure" TLC of order $k+1$.

7 Conclusions and Open Problems

The embeddings of database query languages in the typed λ-calculus that we present here are interesting for a number of reasons:

(1) Our analysis indicates that certain complexity classes can be expressed using fragments of TLC or $\mathrm{TLC}^=$. Can these fragments, under the appropriate input-output conventions, be used to characterize complexity classes exactly? There has been some recent progress on this question. There are a number of functional characterizations of PTIME, e.g., [5, 13, 18, 19, 33]. Another such functional characterization, based on appropriate input-output conventions and order 4 $\mathrm{TLC}^=$ is presented in [23, 24]. Analogous results for PSPACE/EXPTIME and order 5/order 6 $\mathrm{TLC}^=$ are in [23]. These characterizations are proven by an analysis of reduction strategies augmented with data structures. Note that increased order leads to more expressive power as in [26, 29, 31]. Low orders are particularly interesting because of the few primitives in TLC or $\mathrm{TLC}^=$, as opposed to the languages analyzed in [26, 29, 31].

(2) Not all database query languages can be embedded in the typed λ-calculus. Clearly, any computation that is not elementary is not captured by our framework. What should be added to the typed λ-calculus to capture exactly the more powerful database query languages such as the computable queries of [9]?

(3) The use of reduction strategies in PTIME-embeddings provides a link between complexity theory, finite model theory, and the typed λ-calculus. It motivates the further study of reduction strategies in this setting (see [4] for a summary). One interesting problem is the behavior of optimal reduction strategies [34] in our setting—do they yield the same resource bounds as our ad-hoc strategies?

(4) Our PTIME-embeddings identify "pure" functional database query languages without added constructs or added polymorphism. For example, the typed λ-calculus with equality is the simplest syntax to date, that can be used to describe the complex object algebra of [1]. In our embeddings we use lists of tuples and simulate sets by eliminating duplicates. In [1, 6, 7, 28] sets are used as basic constructs, and set iteration is used instead of list iteration. What are the properties of the λ-calculus augmented with set iteration? This is central in the study of database "collection types."

(5) Finally, our embeddings indicate that, at least in the database setting, the case for extensions to TLC should be made on the basis of flexibility and not expressibility. Since the only reasonable queries are the PTIME ones and these are PTIME-embeddable into TLC, it has sufficient expressive power. To make the case for richer type systems, one should examine what algorithms (as opposed to functions) can be described with richer types, but cannot be described in TLC.

Acknowledgment: We would like to thank Catriel Beeri and Serge Abiteboul for many helpful discussions, and two anonymous referees for their insightful comments.

The final version of this manuscript was submitted to the editors after the tragic death of Paris Kanellakis. Gerd Hillebrand and Harry Mairson take this opportunity to acknowledge Paris' contributions to this paper, as well as the greater gifts of his professional insight and personal friendship. It was our privilege to enjoy both in the course of our research collaborations. We submit this paper with great affection for our friend.

References

[1] S. Abiteboul and C. Beeri. *On the Power of Languages for the Manipulation of Complex Objects*. INRIA Research Report 846, 1988.

[2] S. Abiteboul and P. Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *SIGACT News*, **21** (1990), pp. 9–18.

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

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

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

[6] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings DBPL3*, pp. 9–19. Morgan-Kaufmann, 1992.

[7] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Comp. Sci.*, **149** (1995), pp. 3–48.

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

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

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

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

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] Y. Gurevich. Algebras of Feasible Functions. In *Proceedings 24th FOCS* (1983), pp. 210–214.

[20] Y. Gurevich and S. Shelah. Fixed-point Extensions of First Order Logic. *Ann. Pure Appl. Logic*, **32** (1986), pp. 265–280.

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

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

[23] G. Hillebrand. *Finite Model Theory in the Simply Typed Lambda Calculus.* Ph.D. Thesis, Brown Univ., 1994.

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

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

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

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

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

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

[30] P. G. Kolaitis and C. H. Papadimitriou. Why Not Negation By Fixpoint? In *Proceedings 7th PODS*, pp. 231–239, 1988.

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

[32] D. Leivant. Inductive Definitions over Finite Structures. *Information and Computation*, **89** (1990), pp. 95–108.

[33] D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae*, **19** (1993), pp. 167-184.

[34] 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.

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

[36] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Comm. of the ACM*, **3** (1960), pp. 185–195.

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

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

[39] J. Paredaens and D. Van Gucht. Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions. In *Proceedings 7th PODS*, pp. 29–38, 1988.

[40] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[41] J. C. 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 Comp. 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] L. Stockmeyer. The Polynomial-Time Hierarchy. *Theoret. Comp. Sci.*, **3** (1977), pp. 1–22.

[47] J. D. Ullman. *Principles of Database Systems*, 2^{nd} edition. Computer Science Press, 1982.

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