Finite Model Theory in the Simply Typed Lambda Calculus

Gerd G. Hillebrand

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-24
May 1994

Finite Model Theory in the Simply Typed Lambda Calculus

by
Gerd G. Hillebrand
Dipl-Math., WWU Münster, July 1989,
Sc. M., Brown University, May 1991

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1994

This dissertation by Gerd G. Hillebrand
is accepted in its present form by the Department of
Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ______________________________

Paris C. Kanellakis

Recommended to the Graduate Council

Date ______________ ______________________________

Harry G. Mairson

Date ______________ ______________________________

Pascal Van Hentenryck

Approved by the Graduate Council

Date ______________ ______________________________

Vita

I was born on July 6, 1961 in Nordwalde, Germany. After attending high school, I studied mathematics at the Westfälische Wilhelms-Universität in Münster, Germany, from which I graduated in 1989 with a degree of "Diplom-Mathematiker." Then, I joined the Ph.D. program in computer science at Brown University, where I obtained an Sc.M. degree in 1991.

Abstract

Church's simply typed λ-calculus is a very basic framework for functional programming language research. However, it is common to augment this framework with additional programming constructs, because its expressive power for functions over the domain of Church numerals is very limited. In this thesis: (1) We re-examine the expressive power of the "pure" simply typed λ-calculus, but over encodings of finite relational structures, i. e., *finite models* or *databases*. In this novel framework the simply typed λ-calculus expresses all elementary functions from finite models to finite models. In addition, many common database query languages, e. g., relational algebra, Datalog$^{\neg}$, and the Abiteboul/Beeri complex object algebra, can be embedded into it. The embeddings are feasible in the sense that the λ-terms corresponding to PTIME queries can be evaluated in polynomial time. (2) We examine fixed-order fragments of the simply typed λ-calculus to determine machine independent characterizations of complexity classes. For this we augment the calculus with atomic constants and equality among atomic constants. We show that over ordered structures, the order 3, 4, 5, and 6 fragments express exactly the first-order, PTIME, PSPACE, and EXPTIME queries, respectively, and we conjecture that for general $k \geq 1$, order $2\,k+4$ expresses exactly the k-EXPTIME queries and order $2\,k+5$ expresses exactly the k-EXPSPACE queries. (3) We also re-examine other functional characterizations of PTIME and we show that *method schemas with ordered objects* express exactly PTIME. This is a first-order framework proposed for object-oriented databases—as opposed to the above higher-order frameworks. In summary, this research provides a link between finite model theory (and thus computational complexity), database theory, and programming language theory. It sheds new light on the expressive power of the simply typed λ-calculus. It transforms classical complexity questions into questions of expressibility. Finally, it provides new examples of pure functional database query languages.

Credits

Part of the research described in this thesis was done in collaboration with Paris Kanellakis, Harry Mairson, and Sridhar Ramaswamy, and appeared in jointly authored papers. In particular, the material in Chapters 2–5 appeared in [24] and is jointly authored with Paris Kanellakis and Harry Mairson. Part of the material in Chapters 6–7 appeared as a subset of [26]; the results in these chapters are due to the author of this thesis. The material in Chapter 8 appeared in [25] and is jointly authored with Paris Kanellakis and Sridhar Ramaswamy. I am grateful to my co-authors for permission to use parts of our joint work in the present thesis.

While I was at Brown, I was supported by a grant under ONR Contract N00014-91-J-4052, ARPA Order 8225. I gratefully acknowledge this support.

Acknowledgments

First and foremost I would like to thank my advisor, Paris Kanellakis, for "adopting" me when I came to Brown and guiding me through my academic career here. His ability to ask the right questions and make the right connections between seemingly unrelated areas of computer science has sparked both my master's and Ph.D. thesis topics. In particular, it was his observation that simulations of higher order type theory in the lambda calculus could be viewed as queries in the Abiteboul-Beeri complex object calculus that initiated the research carried out in this thesis. In addition to that, he taught me the importance of having JOKES in a talk.

Thanks are also due to his wife, Maria-Teresa Otoya, for her friendship and hospitality and—particularly important for a luxury-deprived graduate student—those excellent dinners.

My thesis committee members Harry Mairson and Pascal Van Hentenryck have taken the time and effort to wade through various more or less inscrutable drafts of the thesis and helped me to make it more presentable. Harry is a coauthor of most of the results, and he has taught me a lot about lambda calculus, functional programming, and the relation between font size and thesis length. Pascal took time out of his sabbatical to read and comment on my thesis in detail, pointing out many interesting connections to functional programming language implementation technology.

I would also like to thank the technical staff of the Brown CS department for providing an exemplary work environment and the administrative staff, in particular Mary Andrade, Katrina Avery, Jennet Kirschenbaum, and Dawn Nicholaus, for helping me in my dealings with the university bureaucracy and in many day-to-day matters.

Finally, I thank my parents, Georg and Magret Hillebrand, who supported me during my undergraduate studies, and my fiancé, Susanne Raab, for her *patience*.

Contents

Chapter 1

Introduction

Motivation and background: Church's simply typed λ-calculus [15] (*typed λ-calculus* for short) is generally considered the "pure" backbone of today's functional programming languages. It is generally felt, however, that it has too little expressive power by itself, and that some sort of polymorphic extension such as the Girard-Reynolds second-order lambda calculus [20, 42] or Milner's ML [40, 41] is necessary to obtain a practically usable language.

The perceived deficiency of the typed λ-calculus stems from various expressibility results for computations over Church numerals. For example, by assuming the inputs and outputs to be Church numerals of type Int, where $\mathsf{Int} \equiv (\tau \to \tau) \to \tau \to \tau$ for some fixed type τ, Schwichtenberg [43] showed that the representable 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 one allows the type τ to be different in each occurrence of Int, then functions like exponentiation or predecessor also become expressible [19]. However, [19] proceeds to show (using a result of Statman) that even under these relaxed conditions, subtraction cannot be expressed.

On the other hand, it is known that provably hard decision problems can be embedded into the typed λ-calculus. For example, Statman [45] shows that deciding whether two typed terms have the same normal form is at least as hard as deciding the truth of a formula in higher-order type theory, a decision problem known to be non-elementary [39]. Mairson's instructive re-proof of Statman's and Meyer's results (in [37]) actually shows how to simulate any elementary-time Turing machine computation using typed λ-terms.

Thus, it appears that the typed λ-calculus can express quite powerful computations, but that the domain of Church numerals is somehow too restrictive to bring this expressive power to bear. This situation is very different from the untyped λ-calculus, which is Turing-complete independent of the particular representation of inputs and outputs chosen.

One way of avoiding the anomalies associated with representations over Church numerals was recently demonstrated by Leivant and Marion [35]. By augmenting the typed lambda calculus with a polymorphic 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 calculus in which exactly the PTIME computations are expressible. However, this calculus makes essential use of the added "impure" features to boost its expressive power.

In this thesis we propose a different avenue. We re-examine the question of representing functions in the "pure" typed λ-calculus, but over *finite first-order relational structures* (*databases* for short) instead of over Church numerals. We represent these structures in a straightforward way, using standard representations of lists and tuples in the typed λ-calculus. For simplicity of presentation, we also equip the typed λ-calculus with an equality over the elements of the domain of the input relational structure. This is not essential for most of our results, though. We show in Section 2.5 how these "impure" features can be simulated in the "pure" calculus.

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*.

Database query languages originated from Codd's work on relational databases [16] and have been primarily studied in the context of finite model theory, e. g., [12, 13, 14, 18, 27, 46, 48]. *Relational calculus/algebra* [16], *Datalog*$^\neg$ [33] and various *fixpoint logics* [5, 13, 14] express practically interesting sets of database queries. In addition, as shown in [27, 48], every PTIME query can be expressed using Datalog$^\neg$ on ordered structures; and, as shown in [5], 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 [3] for a recent survey. Well-known languages in this area are the *complex object calculus/algebra* with or without powerset of [1].

We show in this thesis that all these database query languages can be "naturally embedded" into the typed λ-calculus with equality, and that various complexity classes of queries can be characterized by syntactic properties of their corresponding λ-terms. In this context, "natural embedding" means the following: (1) relations are encoded as lists of tuples, (2) a query is a λ-term which when applied to the encoding of the input relations normalizes to the encoding of the output relation, (3) if a query is computable in polynomial time, then the normal form of the corresponding λ-term can also be computed in polynomial time using a suitable, query-dependent reduction strategy.

Note that our third requirement for natural embeddings is important if one wishes to consider the typed λ-calculus as a reasonable functional database query language. Operators such as *Powerset* in [1] or the second order queries in [13, 18, 46] are powerful, but lead to computational overkill for lower complexity classes, because even low-order polynomial queries such as transitive closure are forced to consume exponential time. The point we make here is that—even though *Powerset* is also expressible in our framework—simple, practical queries can be expressed *efficiently* in this pure functional form, which therefore becomes the canonical functional database language.

The typed λ-calculus, with its syntax, semantics and reduction strategies, should be viewed as a framework for database query languages which is between the declarative calculi and the procedural algebras. After all it is called a calculus, but reductions are procedural. Similar frameworks underlie some practical work in database query languages, e. g., the early FQL language of [11] and the more recent work on structural recursion as a query language [9, 10]. An important difference from [9, 10] is that here we use the pure typed λ-calculus and we use no type polymorphism.

The second contribution of this thesis is a study of the expressive power of fixed-order fragments of the typed λ-calculus. It has long been known that there is some hyper-exponential connection between the order of λ-terms and their expressive power, measured either as the ability to generate long reduction sequences [44], long normal forms [19], or simulate generic Turing machine computations [37]. We show that, unlike the somewhat confusing picture for functions over Church numerals, this connection can be made very precise for computations over finite structures: there is a natural correspondence between fixed-order classes of λ-terms and a hierarchy of query complexity classes ranging from first-order over PTIME, PSPACE, and EXPTIME all the way to elementary time and space. Thus, we obtain new machine-independent characterizations of well-known complexity classes and gain at the same time a better understanding of the expressive power of λ-terms of bounded order. Our proofs here use the constraints on order and input/output behavior of λ-terms representing queries over finite structures to obtain a precise structural analysis of such terms, which facilitates the derivation of upper bounds on the complexity of their evaluation. The reduction strategies we employ here use the order of redexes to decide whether to evaluate them in applicative or normal order.

The final topic of this thesis is a functional query language framework for object-oriented databases (OODBs): *method schemas*. Method schemas [4] are intended to model the object/class/method paradigm of object-oriented programming in the same way that applicative program schemas [17, 21] model traditional programming languages. Method schemas use a simple first-order calculus augmented with key object-oriented features: *classes with methods and inheritance*, a syntax involving *method name overloading*, and a semantics involving *late binding*. In this framework, the higher-order features of the λ-calculus are replaced by named methods, recursion, and late binding. We analyze the expressive power of method schemas over ordered inputs and we show that they express exactly the PTIME queries.

Contributions: The main results in this thesis are: (1) relational algebra, Datalog$^{\neg}$ and the Abiteboul/Beeri complex object algebra can be "naturally embedded" into the typed λ-calculus (with or without equality). (2) Over a suitable representation convention for finite structures, typed terms of order 3 express exactly the first-order queries, typed terms of order 4 express exactly the PTIME queries, typed terms of order 5 express exactly the PSPACE queries and typed terms of order 6 express exactly the EXPTIME queries. We conjecture that for general $k \geq 1$, typed terms of order $2\,k+4$ and $2\,k+5$ express exactly the k-EXPTIME and k-EXPSPACE queries, respectively. (Here, k-EXPTIME stands for time complexity $2^{2^{\cdot^{\cdot^{2^{n^{\mathcal{O}(1)}}}}}}$, where there is a tower of k 2's, and k-EXPSPACE is defined analogously.) (3) PTIME can also be characterized as the set of queries expressible using method schemas over ordered databases.

(1) is joint work with Paris Kanellakis and Harry Mairson; it appeared in [24]. Part of (2) is joint work with Paris Kanellakis; it will appear in [26]. (3) is joint work with Paris Kanellakis and Sridhar Ramaswamy; it appeared in [25].

Overview of this thesis: We begin in Chapter 2 with a description of the main concepts used in this thesis. First, we briefly review the typed λ-calculus with equality in Section 2.1 and its polymorphic variant, core-ML, in Section 2.2. Then, in Section 2.3, we introduce *list iteration*, one of the basic concepts of our framework. List iteration—primitive recursion on lists—was used originally in [37] to re-prove the results of Statman and Meyer mentioned above. It replaces unbounded recursion in the typed λ-calculus, where the type system does not allow a fixed point combinator. We use lists to represent databases as λ-terms; this is detailed in Section 2.4. Finally, we show in Section 2.5 how equality and constants can be simulated in the "pure" typed λ-calculus.

We then proceed with the embedding of the various database query languages, devoting

Chapter 3 to relational algebra, Chapter 4 to inflationary Datalog$^\neg$ (the PTIME queries), and Chapter 5 to the complex object algebra. The proofs here use a progressively refined list iteration technology, which may be of independent interest. One may view it as a "pure" version of simply typed LISP [38]. The requirement that everything be simply typed, without even `let`-polymorphism, complicates some of the proofs (see for example the "type-laundering" technology in Sections 4.2 and 5.3). For all these results we can change the framework to eliminate equality, as described in Section 2.5.

In Chapters 6 and 7, we study query terms of bounded order. For this, we modify the representation conventions of Chapter 2 slightly to minimize the order of encoded data, and we define two families of query languages as fixed-order fragments of the simply typed λ-calculus and core-ML, respectively. By "fine-tuning" the embeddings of Chapters 3–5 to minimize the order, we obtain lower bounds on the expressive power of these query languages. We then proceed in Chapter 7 with the design of efficient evaluation strategies for these languages, which enable us to obtain matching upper bounds on their expressive power. Note that, here equality is an integral part of the setting; it is open whether similar bounds can be obtained without it.

In Chapter 8, we shift our attention to object-oriented database query languages by introducing method schemas and analyzing their expressive power over ordered inputs. We first describe the syntax and semantics of method schemas in Sections 8.1 and 8.2. Then, in Section 8.3, we show that method schemas over ordered databases express exactly the PTIME queries, by showing that method schemas and Datalog$^\neg$ can be embedded into each other.

We conclude in Chapter 9 with some directions for future research.

Chapter 2

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

2.1 Basic Definitions

The syntax of *types* is given by the grammar $\mathcal{T} \equiv t \mid (\mathcal{T} \rightarrow \mathcal{T})$, where t ranges over a countable set of *type variables* $\{\rho, \sigma, \tau, \ldots\}$. Thus, ρ is a type, as are $(\rho \rightarrow \sigma)$ and $(\rho \rightarrow (\rho \rightarrow \rho))$. We denote types by lowercase Greek letters $\alpha, \beta, \gamma, \ldots$.

Typed λ-terms are given by the grammar $\mathcal{E} \equiv \xi \mid (\mathcal{E}\mathcal{E}) \mid \lambda\xi{:}\,\mathcal{T}.\,\mathcal{E}$, where ξ ranges over a countable set of *expression variables* $\{x, y, z, \ldots\}$. We denote typed λ-terms by lowercase Latin letters $e, f, g, \ldots$. To keep our notation simple, we adopt the standard associativity conventions for types and terms: the type $\alpha \rightarrow \beta \rightarrow \gamma$ stands for $\alpha \rightarrow (\beta \rightarrow \gamma)$ and the term $e\,f\,g$ stands for $(e\,f)\,g$.

Well-typedness 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' augmenting Γ with $\Gamma'(x) = \alpha$:

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

(Abs) $$\frac{\Gamma\,[x{:}\,\alpha] \vdash e{:}\,\beta}{\Gamma \vdash \lambda x{:}\,\alpha.\,e : \alpha \rightarrow \beta}$$

(App) $$\frac{\Gamma \vdash e{:}\,\alpha \rightarrow \beta \quad \Gamma \vdash e'{:}\,\alpha}{\Gamma \vdash e\,e'{:}\,\beta}$$

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

The *order* of a type, which measures the higher-order functionality of a λ-term of that type, is defined as $\mathrm{order}(\tau) = 0$ for a type variable τ, and $\mathrm{order}(\alpha \to \beta) = \max(1 + \mathrm{order}(\alpha), \mathrm{order}(\beta))$. We also refer to the order of a typed λ-term as the order of its type.

For typed λ-terms e, e', we write $e \rhd_\alpha e'$ (α-reduction) when e' can be derived from e by renaming of a λ-bound variable, for example $\lambda x{:}\gamma.\,\lambda y{:}\delta.\,y \rhd_\alpha \lambda x{:}\gamma.\,\lambda z{:}\delta.\,z$. We write $e \rhd_\beta e'$ (β-reduction) when e' can be derived from e by replacing a subterm in e of the form $(\lambda x{:}\gamma.\,f)\,g$ by $f\,[x{:}= g]$ (f with g substituted for all free occurrences of x in f). Reduction preserves types [7, Theorem 4.2.8]. We write $\rhd$ for the reflexive, transitive closure of $\rhd_\alpha$ and $\rhd_\beta$.

In this paper, we consider a mild modification of the standard presentation above, by enriching the simply-typed λ-calculus with a countably infinite set $\{o_1, o_2, \ldots\}$ of constants of type o, and for some fixed type α, introducing an equality constant $Eq{:}\,\mathsf{o} \to \mathsf{o} \to \alpha \to \alpha \to \alpha$. For every pair of constants $o_i, o_j{:}\,\mathsf{o}$, we add to $\rhd$ the reduction rule

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

These are known as delta reductions. Thus, if e and f are terms of type α, we have $(Eq\, o_i\, o_j\, e\, f) \rhd e$ if $i = j$ and $(Eq\, o_i\, o_j\, e\, f) \rhd f$ otherwise. The lack of polymorphism in the simply typed λ-calculus requires that α be fixed once and for all; we will see later that for our purposes, it suffices to choose $\alpha{:}= \tau$, where τ is a fixed type variable.

In Section 2.5 below, we observe that by changing the input-output conventions it is possible to obtain an embedding in the pure typed λ-calculus without constants and equality.

The modified system enjoys the following well-known properties:

Theorem 2.1.1 (Church-Rosser property) *If $e \rhd e'$ and $e \rhd e''$, then there exists a λ-term e''' such that $e' \rhd e'''$ and $e'' \rhd e'''$.*

Theorem 2.1.2 (Strong normalization property) *For each e, there exists an integer n such that if $e \rhd e'$, then the derivation involves no more than n β-reductions.*

For more details, see for example [15, 6]. In particular, [6, Section 15.3] discusses the Church-Rosser property for the untyped λ-calculus enriched with equality.

2.2 Let-Polymorphism: Core-ML

Core-ML: The syntax of core-ML is the syntax of TLC augmented with one new expression construct: $\mathcal{E} \equiv x \mid (\mathcal{E}\mathcal{E}) \mid \lambda x.\mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ \mathcal{E}$. The simplest way of explaining

ML types involves the same monomorphic types and rules (Var), (Abs), and (App) used for TLC with one additional rule that captures the polymorphism (see [31]):

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

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

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

The analogous definitions, expressibility, principal type and type inference properties hold for core-ML$^=$, where constants and their equality are added as in TLC$^=$. Order of functionality is defined in the same way. There are two differences: (1) Type inference is no longer in linear time but EXPTIME-complete [31, 32]. (2) Arbitrary order core-ML, core-ML$^=$, TLC, and TLC$^=$ all have the same expressive power, but for fixed order type inference allows more core-ML than TLC programs to be typed, so it might provide more expressibility.

2.3 List Iteration

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

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

is a λ-term of type $(\alpha \to \beta \to \beta) \to \beta \to \beta$, for *any* type β—in other words, L is a typable term no matter what type β we choose (though one fixed type must be chosen). 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,

[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.

starting from the last and proceeding backwards to the first. 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 the expression $\mathit{Xor}\,e_1\,(\mathit{Xor}\,e_2 \ldots (\mathit{Xor}\,e_k\,\mathit{False}) \ldots)$, 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 can be coded as a term of type $\mathcal{B} \equiv (\mathsf{Bool} \to \sigma \to \sigma) \to \sigma \to \sigma$ for any type σ. To type $\mathit{Parity}\,L$, we must type L with $\mathsf{Bool} \equiv \tau \to \tau \to \tau$ substituted for σ, which we write as $\mathcal{B}\,[\sigma{:=}\,\mathsf{Bool}]$. Similarly, to type $\mathit{Length}\,L$, we need to type $L{:}\,\mathcal{B}\,[\sigma{:=}\,\mathsf{Int}]$. Thinking of the type σ of L as an output type *variable*, in both instances we see that the output type must be "contaminated" or "raised" to provide the primitive recursive iterator. If we want the output type of input L to be fixed, and two different output types are needed to carry out the computation, this poses a problem.

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

We point out that problem (2) can sometimes be handled by alternate encodings, for example $\mathit{Length} \equiv \lambda L{:}(\alpha \to \sigma \to \sigma) \to \sigma \to \sigma.\, \lambda s{:}\sigma \to \sigma.\, \lambda z{:}\sigma.\, L\,(\lambda x{:}\alpha.\, s)\, z$, where L is used to iterate over objects of lower type. Problem (1) can be solved by defining a "type-laundering" operator that transforms a "contaminated" list, e. g., $L{:}\mathcal{B}\,[\sigma{:=}\,\mathsf{Int}]$, into a "clean" list of type just $\mathcal{B}$. Both techniques, with elaborate variants, are exploited repeatedly in the following sections.

2.4 Databases as λ-Terms

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

Let $\{o_1, o_2, \ldots\}$ be the set of constants of the TLC$^=$ calculus. For convenience, we assume that the same set of constants also serves as the universe over which all structures are defined. Furthermore, we fix, once and for all, a type variable τ.

Boolean values are represented by

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

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

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

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

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 $1 \le i \le k$, as α^k. In particular, o^k is the type of a k-tuple of constants.

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

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

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

Relations are represented as duplicate-free lists of tuples of constants. Thus, the type of a k-ary relation is $\{\mathsf{o}^k\}$. All our encodings make sure that their output is duplicate-free, provided that the input is duplicate-free.

Queries are represented as typed λ-terms of the form $Q \equiv \lambda R_1 \ldots \lambda R_l.\, M$, which when applied to encodings $\overline{r_1}, \ldots, \overline{r_k}$ of input relations, reduce to a normal form which is the encoding of the desired output relation. We measure the time complexity of computation by the length of reduction sequences, according to a suitable reduction strategy, as a function of the database size. Our choice of reduction strategies is such that the size of intermediate terms is always polynomial in the size of the input (except where *Powerset* is involved). Hence the time needed to perform a reduction sequence on a sequential machine is within a polynomial factor of the length of the reduction sequence.

Note that the encoding of a relation implicitly provides an ordering of the database universe. It is possible to write queries whose outcome depends on the ordering of the input. This seems inherent in our framework, since the λ-calculus does not provide (unordered) sets as primitive objects. However, all the encodings given below are order-independent, i. e., generic.

2.5 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 (see [37] for another way of coding up equality on finite domains).

Let $O = \{o_1, o_2, \ldots, o_N\}$ be the set of constants occurring in a particular input. We can code the constant o_i as the projection function

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

and the equality predicate as

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

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 explicit constants are replaced by their corresponding projection functions. Note that the encoding of *Eq* now depends on the size of the input universe; so in this setting, *Eq* has to be part of the input. It is easy to see that the query terms described in the following sections work unchanged in the "pure" setting, except that the symbol *Eq* has to be λ-bound at the outermost level.

Chapter 3

Coding Relational Algebra

We begin by demonstrating how the relational algebra operators of [16] can be represented in the simply typed λ-calculus with equality. Every operator is coded as a λ-term that takes one or two relations in the list-of-tuples format described in Section 2.4 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.

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.

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

Here, Eq denotes the equality predicate for constants, of type $\mathsf{o} \to \mathsf{o} \to \mathsf{Bool}$. Once t and s 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 $(Eq\, x_i\, y_i)$ terms reduce to either $\mathit{True} \equiv \lambda u.\, \lambda v.\, u$ or $\mathit{False} \equiv \lambda u.\, \lambda v.\, v$. Hence the "body"

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

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 $(\mathit{Equal}_k\, t\, s)$. First, t and s are reduced to normal form. If either t or s does not reduce to 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 for the query terms presented below). Otherwise, the normal forms of t and s are substituted into the body of the Equal_k predicate and the topmost 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.

The following term checks whether a k-tuple $\lambda f.\, f\, o_{i_1} \ldots o_{i_k}$ occurs in a list $\lambda c.\, \lambda n.\, c\, t_1\, (c\, t_2 \ldots (c\, t_m\, n) \ldots)$ of k-tuples by comparing the tuple to every element of the list.

$$
\begin{array}{l}
\mathit{Member}_k{:}\, \mathsf{o}^k \to \{\mathsf{o}^k\} \to \mathsf{Bool} \equiv \\
\quad \lambda t{:}\, \mathsf{o}^k.\, \lambda R{:}\, \{\mathsf{o}^k\}. \\
\quad\quad \lambda u{:}\, \tau.\, \lambda v{:}\, \tau. \\
\quad\quad\quad R\, (\lambda r{:}\mathsf{o}^k.\, \lambda T{:}\, \tau.\, (\mathit{Equal}_k\, t\, r)\, u\, T)\, v
\end{array}
$$

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 *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: First, t and R are reduced to normal form. If this does not yield the encoding of a tuple and a relation, respectively, no further reductions are performed (again, we will see that this case does not occur for the terms presented below). Otherwise, the normal forms of t and R are substituted into the body of the *Member* predicate and a copy of the "loop body" $(\mathit{Equal}_k\, t\, r)\, u\, T$ is evaluated once for every tuple in R, beginning with the last tuple and proceedings backwards to the first. At each step, the current tuple of R is substituted for r and the result of the previous iteration (initially v) is substituted for T. Since both arguments to *Equal* are then 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 T in order to select the value to pass on to the next stage.

With this reduction strategy, the final result can be determined in $O(|R|)$ reduction steps, where $|R|$ is the size of the relation encoded by R. The same bound holds on 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 S, we build a new relation T by "walking down" R and testing each tuple for membership in S. If the tuple occurs in S, it is included in the output, otherwise it is ignored.

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

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

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

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 $(\mathit{Intersection}\, R\, S)$—and, with the obvious modifications, for the next two relational operators as well—is as follows: First, R and S

are reduced to normal form and nothing further is done if this does not yield encodings of relations. Otherwise, the normal forms are substituted into the body of the *Intersection* operator and the "loop body" $(Member_k\, r\, S)\,(c\, r\, T)\, T$ is evaluated once for each tuple in R, from last to first, with the current tuple substituted for r and the result of the previous iteration (initially n) substituted for T. With the reduction strategy for *Member* outlined above, each stage of the iteration requires $O\,(|S|)$ reduction steps. Hence, the final result can be computed in $O\,(|R||S|)$ steps, using intermediate terms of size at most $O\,(|R||S|)$.

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

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

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

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

(If we knew that R and S were disjoint, we could compute their union more simply as $\lambda c.\, \lambda n.\, R\, c\, (S\, c\, n)$. In the general case, however, we need the more complex term above to ensure a duplicate-free output.)

Having dealt with the "set-oriented" operators, we now examine the "tuple-oriented" operators. We need two auxiliary terms first: The $Concat_{k,l}$ operator concatenates a k-tuple and an l-tuple to form a $(k+l)$-tuple, and the $Rearrange_{k;i_1,\ldots,i_l}$ 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.

$$
\begin{aligned}
&Concat_{k,l} \colon \mathsf{o}^k \to \mathsf{o}^l \to \mathsf{o}^{k+l} \equiv \\
&\quad \lambda t\colon \mathsf{o}^k.\, \lambda s\colon \mathsf{o}^l.
\end{aligned}
$$

$$
\begin{array}{l}
\quad\lambda f{:}\,\mathsf{o} \to \cdots \to \mathsf{o} \to \tau. \\
\quad\quad t\,(\lambda x_1{:}\,\mathsf{o} \ldots \lambda x_k{:}\,\mathsf{o}. \\
\quad\quad\quad s\,(\lambda y_1{:}\,\mathsf{o} \ldots \lambda y_l{:}\,\mathsf{o}. \\
\quad\quad\quad\quad f\,x_1 \ldots x_k\,y_1 \ldots y_l))
\end{array}
$$

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

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

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

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

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

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

The reduction strategy for the *Times* operator follows the now familiar pattern: First, the arguments are reduced to normal form and nothing further is done if this does not lead to encodings of relations. Otherwise, the normal forms are substituted into the body of the operator and the "outer loop body" $S\,(\lambda s.\, \lambda U.\, c\,(\mathit{Concat}\,r\,s)\,U)\,T$ is evaluated once for each tuple in R, from last to first, with r and T 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" $c\,(\mathit{Concat}\, r\, s)\, U$ once for each tuple in S, which eventually leads to an evaluation of $(\mathit{Concat}\, r\, s)$ once for every combination of tuples $r \in R$ and $s \in S$. It is easy to see that the entire procedure takes $O\,(|R||S|)$ reduction steps and uses $O\,(|R||S|)$ space for intermediate terms.

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

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

The reduction strategy for *Select* again consists of reducing the argument first and then evaluating the "loop body" once for each tuple in R. Reduction sequence length and term size are bounded by $O\,(|R|)$.

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

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

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

$$
\begin{aligned}
&\mathit{First}_{k;i_1,\ldots,i_l}\colon \mathsf{o}^k \to \{\mathsf{o}^k\} \to \mathsf{Bool} \equiv\\
&\quad \lambda t\colon \mathsf{o}^k.\, \lambda R\colon \{\mathsf{o}^k\}.
\end{aligned}
$$

$$
\begin{array}{l}
\lambda u\colon\tau.\,\lambda v\colon\tau.\\
\quad R\,(\lambda r\colon \mathsf{o}^k.\,\lambda T\colon\tau.\\
\qquad \mathit{Equal}_l\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\,t)\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\,r)\,(\mathit{Equal}_k\,t\,r\,u\,v)\,T)\,u
\end{array}
$$

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

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

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

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

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:

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

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

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

and then, using 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 $O\,(|R|^2)$.

This completes the coding of relational algebra. It is straightforward to verify that every term given above is well-typed and, when applied to one (resp., two) relations coded in the format described in Section 2.4, reduces to a normal form which is the encoding of the desired output relation. Moreover, when the terms are reduced according to the specified reduction strategies, 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 3.0.1 *Let $\phi(R_1, \ldots, R_l)$ be a relational algebra expression over relational schema $\mathcal{S} = (R_1, \ldots, R_l)$. Then there exists a term ψ of the simply typed λ-calculus with equality such that for every instance $(r_1, \ldots, r_l)$ of $\mathcal{S}$, the expression $(\psi\, \overline{r_1} \ldots \overline{r_l})$ reduces to $\overline{\phi(r_1, \ldots, r_l)}$, where $\overline{r}$ denotes the encoding of relation r as described in Section 2.4. Moreover, the normal form of $(\psi\, \overline{r_1} \ldots \overline{r_l})$ can be computed in time polynomial in the size of $r_1, \ldots, r_l$.*

Chapter 4

Coding Fixpoint Queries

4.1 Line of Attack

With the machinery of the previous section at our disposal, we can now code arbitrary relational queries as typed λ-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].

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 database universe. 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 D be a TLC$^=$ terms which computes the input universe, i. e., the set of constants occurring in the relations encoded by $R_1, \ldots, R_l$. D can be computed as the union of all columns of $R_1, \ldots, R_l$:

$D\colon \{\mathsf{o}\} \equiv$

$$
\begin{array}{l}
(\mathit{Union}_1\,(\mathit{Project}_{k_1;1}\,R_1)\,(\mathit{Union}_1\,(\mathit{Project}_{k_1;2}\,R_1)\ldots \\
\quad(\mathit{Union}_1\,(\mathit{Project}_{k_2;1}\,R_2)\,(\mathit{Union}_1\,(\mathit{Project}_{k_2;2}\,R_2)\ldots \\
\qquad\ldots \\
\qquad\quad(\mathit{Union}_1\,(\mathit{Project}_{k_l;1}\,R_l)\,(\mathit{Union}_1\,(\mathit{Project}_{k_l;2}\,R_l)\ldots)
\end{array}
$$

Since the *Union* and *Project* operators eliminate duplicates, the length of D is exactly n, the size of the database universe. To obtain an iterator of length n^k, we form the k-fold Cartesian product of D with itself:

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

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

$$\mathit{Fix}_\phi \equiv \lambda R_1\ldots\lambda R_l.\,\mathit{Crank}\,(\lambda t.\,\lambda R.\,Q)\,\mathit{Nil}.$$

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 and in polynomial space.

4.2 Fixing the Types

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 $\{\mathsf{o}^{k_i}\} \equiv (\mathsf{o}^{k_i} \to \tau \to \tau) \to \tau \to \tau$. 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 $(\mathsf{o}^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}$. 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 $\mathit{Copy}_{k_i,k}$ which takes an encoding of a relation r_i of type $(\mathsf{o}^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}$ and produces another encoding of r_i of type $(\mathsf{o}^{k_i} \to \tau \to \tau) \to \tau \to \tau$. That is, $\mathit{Copy}_{k_i,k}$ computes the identity function on encodings of relations, but forces different types for its input and output.

$\mathit{Copy}_{k_i,k}$ can be written as follows:

$$\begin{aligned}
&\mathit{Copy}_{k_i,k}\colon ((\mathsf{o}^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^{k_i}\} \equiv \\
&\quad \lambda R\colon (\mathsf{o}^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}. \\
&\qquad \lambda c\colon \mathsf{o}^{k_i} \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad R\ (\lambda r\colon \mathsf{o}^{k_i}.\, \lambda T\colon \{\mathsf{o}^k\}.\, \lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, c\, r\, (T\, c'\, n')) \\
&\qquad\qquad (\lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, n) \\
&\qquad\qquad (\lambda s\colon \mathsf{o}^k.\, \lambda S\colon \tau.\, S)\, n
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{SimpleCopy}\colon \{\mathsf{o}^{k_i}\} \to \{\mathsf{o}^{k_i}\} \equiv \\
&\quad \lambda R\colon \{\mathsf{o}^{k_i}\}. \\
&\qquad \lambda c\colon \mathsf{o}^{k_i} \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad R\, (\lambda r\colon \mathsf{o}^{k_i}.\, \lambda T\colon \tau.\, c\, r\, T)\, n
\end{aligned}$$

Clearly, this term computes the identity function on encodings of relations. If we now constrain the type of the input to be $(\mathsf{o}^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}$, then the type of the "loop body" $(\lambda r\colon \mathsf{o}^{k_i}.\, \lambda T\colon \tau.\, c\, r\, T)$, which used to be $\mathsf{o}^{k_i} \to \tau \to \tau$, must become $\mathsf{o}^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}$. The easiest way of doing this is by prepending dummy λ-abstractions to the value computed at each step and removing them again at the next stage by applying the incoming value to dummy arguments. This leads to a "loop body"

$$(\lambda r\colon \mathsf{o}^{k_i}.\, \lambda T\colon \{\mathsf{o}^k\}.\, \lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, c\, r\, (T\, c'\, n')),$$

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

$$\begin{aligned}
&R\ (\lambda r\colon \mathsf{o}^{k_i}.\, \lambda T\colon \{\mathsf{o}^k\}.\, \lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, c\, r\, (T\, c'\, n')) \\
&\quad (\lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, n).
\end{aligned}$$

That almost works: if R is a list

$$\lambda c\colon o^{k_i} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}.\, \lambda n\colon \{\mathsf{o}^k\}.\, c\, t_1\, (c\, t_2 \ldots (c\, t_m\, n) \ldots),$$

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

$$\lambda c'\colon o^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, c\, t_1\, (c\, t_2 \ldots (c\, t_m\, n) \ldots).$$

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

$$
\begin{aligned}
&\lambda c\colon \mathsf{o}^{k_i} \to \tau \to \tau.\, \lambda n\colon \tau.\\
&\quad R\,(\lambda r\colon \mathsf{o}^{k_i}.\, \lambda T\colon \{\mathsf{o}^k\}.\, \lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, c\, r\, (T\, c'\, n'))\\
&\qquad (\lambda c'\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n'\colon \tau.\, n)\\
&\qquad (\lambda s\colon \mathsf{o}^k.\, \lambda S\colon \tau.\, S)\, n
\end{aligned}
$$

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

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

$$\mathit{Fix}_\phi \equiv \lambda R_1 \ldots \lambda R_l.\, \mathit{Crank}\,(\lambda t.\, \lambda R.\, Q)\, \mathit{Nil}$$

by replacing every occurrence of R_i in Q by $(\mathit{Copy}_{k_i,k}\, R_i)$, i. e., by writing

$$
\begin{aligned}
&\mathit{Fix}_\phi \equiv\\
&\quad \lambda R_1\colon (\mathsf{o}^{k_1} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}.\\
&\quad \ldots\\
&\quad \lambda R_l\colon (\mathsf{o}^{k_l} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}) \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\}.\\
&\qquad \mathit{Crank}\,(\lambda t\colon \mathsf{o}^k.\, \lambda R\colon \{\mathsf{o}^k\}.\, Q\,[R_1\colon = (\mathit{Copy}_{k_1,k}\, R_1), \ldots, R_l\colon = (\mathit{Copy}_{k_l,k}\, R_l)])\, \mathit{Nil}.
\end{aligned}
$$

It is straightforward, though tedious, to verify that this is indeed a well-typed term. Since $(\mathit{Copy}_{k_i,k}\, R_i)$ 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 $(\mathit{Copy}_{k_i,k}\, R_i)$ only once. Hence, we have the following theorem:

Theorem 4.2.1 *Let $(\mu_R\, \phi)$ be a fixpoint query over the relational schema $\mathcal{S} = (R_1, \ldots, R_l)$. Then there exists a term Fix_ϕ of the simply typed λ-calculus with equality such that for every instance $(r_1, \ldots, r_l)$ of S, the expression $(\mathrm{Fix}_\phi\, \overline{r_1} \ldots \overline{r_l})$ reduces to $\overline{(\mu_R\, \phi)\,(r_1, \ldots, r_l)}$, where $\overline{r}$ denotes the encoding of relation r as described in Section 2.4. Moreover, the normal form of $(\mathrm{Fix}_\phi\, \overline{r_1} \ldots \overline{r_l})$ can be computed in time polynomial in the size of $r_1, \ldots, r_l$.*

4.3 An Example: Transitive Closure

To illustrate the concepts of the previous sections, 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 Section 4.2, the translation into TLC$^=$ of this fixpoint query is (we omit type annotations):

$$\begin{array}{l} TC \equiv \lambda R_1. \\ \qquad\qquad \mathit{Crank} \\ \qquad\qquad\quad (\lambda t.\, \lambda R. \\ \qquad\qquad\qquad \mathit{Union}_2\,(\mathit{Copy}_{2,2}\,R_1)\,(\mathit{Project}_{4;1,4}\,(\mathit{Select}_{4;2=3}\,(\mathit{Times}_{2,2}\,R\,R)))) \\ \qquad\qquad\quad \mathit{Nil}, \end{array}$$

where

$$\mathit{Crank} \equiv (\mathit{Times}_{1,1}\,D\,D)$$

and

$$D \equiv \mathit{Union}_1\,(\mathit{Project}_{2;1}\,R_1)\,(\mathit{Project}_{2;2}\,R_1).$$

Assume that R_1 encodes the relation $[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]$. According to the reduction strategy for fixpoint queries, we first have to evaluate the expressions Crank and $(\mathit{Copy}_{2,2}\,R_1)$. For the latter, we have (omitting some intermediate steps)

$$\begin{array}{l} (\mathit{Copy}_{2,2}\,R_1) \\ \triangleright\ \lambda c.\,\lambda n. \\ \qquad (\lambda r.\,\lambda T.\,\lambda c'.\,\lambda n'.\,c\,r\,(T\,c'\,n'))\,\langle 1,2\rangle \\ \qquad\quad ((\lambda r.\,\lambda T.\,\lambda c'.\,\lambda n'.\,c\,r\,(T\,c'\,n'))\,\langle 2,3\rangle \\ \qquad\qquad ((\lambda r.\,\lambda T.\,\lambda c'.\,\lambda n'.\,c\,r\,(T\,c'\,n'))\,\langle 3,4\rangle \\ \qquad\qquad\quad (\lambda c'.\,\lambda n'.\,n)))\,(\lambda s.\,\lambda S.\,S)\,n \end{array}$$

$$
\begin{aligned}
&\rhd\ \lambda c.\, \lambda n.\\
&\quad (\lambda c'.\, \lambda n'.\\
&\qquad c\, \langle 1,2\rangle\\
&\qquad\quad (c\, \langle 2,3\rangle\\
&\qquad\qquad (c\, \langle 3,4\rangle\, n)))\, (\lambda s.\, \lambda S.\, S)\, n\\
&\rhd\ \lambda c.\, \lambda n.\, c\, \langle 1,2\rangle\, (c\, \langle 2,3\rangle\, (c\, \langle 3,4\rangle\, n))\\
&\equiv\ [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle],
\end{aligned}
$$

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

$$D \equiv \mathit{Union}_1\, (\mathit{Project}_{2;1}\, R_1)\, (\mathit{Project}_{2;2}\, R_1).$$

We illustrate the reduction sequence for $(\mathit{Project}_{2;1}\, R_1)$, according to the reduction strategy for projection:

$$
\begin{aligned}
&(\mathit{Project}_{2;1}\, R_1)\\
&\rhd\ \lambda c.\, \lambda n.\\
&\quad (\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 1,2\rangle\\
&\qquad ((\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 2,3\rangle\\
&\qquad\quad ((\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 3,4\rangle\, n))\\
&\rhd\ \lambda c.\, \lambda n.\\
&\quad (\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 1,2\rangle\\
&\qquad ((\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 2,3\rangle\\
&\qquad\quad ((\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))\\
&\rhd\ \lambda c.\, \lambda n.\\
&\quad (\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 1,2\rangle\\
&\qquad ((\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 2,3\rangle\\
&\qquad\quad (\mathit{True}\, (c\, (\mathit{Rearrange}_{2;1}\, \langle 3,4\rangle)\, n)\, n))\\
&\rhd\ \lambda c.\, \lambda n.\\
&\quad (\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 1,2\rangle\\
&\qquad ((\lambda r.\, \lambda T.\, (\mathit{First}_{2;1}\, r\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle])\, (c\, (\mathit{Rearrange}_{2;1}\, r)\, T)\, T)\, \langle 2,3\rangle
\end{aligned}
$$

$$(c\,\langle 3\rangle\,n))$$

$$\rhd \cdots$$

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

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

$(Project_{2;2}\,R_1)$ reduces to $[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle]$ and thus D reduces as follows:

$$D \equiv\ Union_1\,[\langle 1\rangle, \langle 2\rangle, \langle 3\rangle]\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle]$$

$$\begin{aligned}
\rhd\ & \lambda c.\,\lambda n. \\
& (\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 1\rangle \\
& \quad ((\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 2\rangle \\
& \quad\quad ((\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 3\rangle \\
& \quad\quad\quad (c\,\langle 2\rangle\,(c\,\langle 3\rangle\,(c\,\langle 4\rangle\,n)))))
\end{aligned}$$

$$\begin{aligned}
\rhd\ & \lambda c.\,\lambda n. \\
& (\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 1\rangle \\
& \quad ((\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 2\rangle \\
& \quad\quad ((Member_1\,\langle 3\rangle\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle]) \\
& \quad\quad\quad (c\,\langle 2\rangle\,(c\,\langle 3\rangle\,(c\,\langle 4\rangle\,n))) \\
& \quad\quad\quad (c\,\langle 3\rangle\,(c\,\langle 2\rangle\,(c\,\langle 3\rangle\,(c\,\langle 4\rangle\,n))))))
\end{aligned}$$

$$\begin{aligned}
\rhd\ & \lambda c.\,\lambda n. \\
& (\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 1\rangle \\
& \quad ((\lambda r.\,\lambda T.\,(Member_1\,r\,[\langle 2\rangle, \langle 3\rangle, \langle 4\rangle])\,T\,(c\,r\,T))\,\langle 2\rangle \\
& \quad\quad\quad (c\,\langle 2\rangle\,(c\,\langle 3\rangle\,(c\,\langle 4\rangle\,n))))
\end{aligned}$$

$$\rhd \cdots$$

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

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

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

$$(\lambda t.\,\lambda R.\,Union_2\,[\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]\,(Project_{4;1,4}\,(Select_{4;2=3}\,(Times_{2,2}\,R\,R))))\,\langle 1,1\rangle$$

$$((\lambda t.\, \lambda R.\, Union_2\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]\, (Project_{4;1,4}\, (Select_{4;2=3}\, (Times_{2,2}\, R\, R))))\, \langle 1,2\rangle$$

$$\ldots$$

$$((\lambda t.\, \lambda R.\, Union_2\, [\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle]\, (Project_{4;1,4}\, (Select_{4;2=3}\, (Times_{2,2}\, R\, R))))\, \langle 4,4\rangle$$

$$(\lambda c.\, \lambda n.\, n)))$$

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

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

Chapter 5

The Complex Object Algebra

5.1 Basic Definitions

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.

Let us briefly describe the salient features of the complex object model (for a formal definition, see [3]).

Complex objects[1] are built from constants using tuple ($\langle\rangle$) and (finite) set ($\{\}$) constructors. Thus, 1 is a complex object, as are $\langle 1,2\rangle$, $\{1,2,3\}$, and $\{1,\langle 1,2\rangle,\{1,2,3\}\}$. We encode complex objects in the natural way, using λ-calculus constants at the bottom level and then combining them into tuples and lists as described in Section 2.4.

Relations are sets of objects of a common type (not necessarily flat). For example, $\{\langle 1,\{2,3\}\rangle,\langle 4,\{5,6\}\rangle\}$ is a relation. A relation may be viewed as a single complex object of type set, and it is encoded as such.

Algebra operations map relations to relations and include the conventional relational algebra operations plus the following:

- *Powerset* (of type $\{\alpha\} \to \{\{\alpha\}\}$), defined by $\mathit{Powerset}(R) := \{S \mid S \subseteq R\}$,

[1] The term "complex value" might be more appropriate, since we do not consider object identifiers or methods. However, the name "complex object algebra" seems to be firmly established.

- *Set* (of type $\{\alpha\} \to \{\{\alpha\}\}$), defined by $\mathit{Set}(R) := \{\{x\} \mid x \in R\}$.
- *Setcomb* (of type $\{\{\alpha\}\} \to \{\alpha\}$), defined by $\mathit{Setcomb}(R) := \cup_{S \in R} S$.
- *Tup* (of type $\{\alpha\} \to \{\alpha^1\}$), defined by $\mathit{Tup}(R) := \{\langle x \rangle \mid x \in R\}$.
- *Tupcomb* (of type $\{\alpha^1\} \to \{\alpha\}$), defined by $\mathit{Tupcomb}(R) := \{x \mid \langle x \rangle \in R\}$.

In addition, the *Select* operator is extended to allow selection conditions of the form $x_i = c$, $x_i = x_j$, $x_i \in x_j$, and $x_i = x_j.d$, where $x_1, \ldots, x_k$ are the fields of the tuple under consideration and $x_j.d$ denotes the d^{th} field of x_j if x_j is itself a tuple.

5.2 Coding the Operators

As it turns out, the implementation of the relational algebra operators described in Chapter 3 works with virtually 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_α, Member_α, and $\mathit{Subset}_{\{\alpha\}}$ that encode the corresponding predicate for objects of that type, using the already defined encodings for objects of less complex type.

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

$$
\begin{aligned}
&\mathit{Equal}_{\{\alpha\}} : \{\alpha\} \to \{\alpha\} \to \mathsf{Bool} \equiv \\
&\quad \lambda R{:}\,\{\alpha\}.\, \lambda S{:}\,\{\alpha\}. \\
&\qquad \lambda u{:}\,\tau.\, \lambda v{:}\,\tau. \\
&\qquad\quad (\mathit{Subset}_{\{\alpha\}}\; R\; S)\,((\mathit{Subset}_{\{\alpha\}}\; S\; R)\; u\; v)\; v
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Equal}_{\alpha_1 \times \cdots \times \alpha_k} : \alpha_1 \times \cdots \times \alpha_k \to \alpha_1 \times \cdots \times \alpha_k \to \mathsf{Bool} \equiv \\
&\quad \lambda t{:}\,\alpha_1 \times \cdots \times \alpha_k.\, \lambda s{:}\,\alpha_1 \times \cdots \times \alpha_k. \\
&\qquad \lambda u{:}\,\tau.\, \lambda v{:}\,\tau.
\end{aligned}
$$

$$
\begin{array}{l}
t\,(\lambda x_1{:}\,\alpha_1 \ldots \lambda x_k{:}\,\alpha_k.\\
\quad s\,(\lambda y_1{:}\,\alpha_1 \ldots \lambda y_k{:}\,\alpha_k.\\
\qquad ((\mathit{Equal}_{\alpha_1}\, x_1\, y_1)\\
\qquad\quad \ldots\\
\qquad\qquad ((\mathit{Equal}_{\alpha_k}\, x_k\, y_k)\, u\, v)\, v) \ldots v)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Member}_{\alpha}{:}\, \alpha \to \{\alpha\} \to \mathsf{Bool} \equiv\\
\quad \lambda t{:}\, \alpha.\, \lambda R{:}\, \{\alpha\}.\\
\qquad \lambda u{:}\, \tau.\, \lambda v{:}\, \tau.\\
\qquad\quad R\,(\lambda r{:}\, \alpha.\, \lambda T{:}\, \tau.\, (\mathit{Equal}_{\alpha}\, t\, r)\, u\, T)\, v
\end{array}
$$

$$
\begin{array}{l}
\mathit{Subset}_{\{\alpha\}}{:}\, \{\alpha\} \to \{\alpha\} \to \mathsf{Bool} \equiv\\
\quad \lambda R{:}\, \{\alpha\}.\, \lambda S{:}\, \{\alpha\}.\\
\qquad \lambda u{:}\, \tau.\, \lambda v{:}\, \tau.\\
\qquad\quad R\,(\lambda r{:}\, \alpha.\, \lambda T{:}\, \tau.\, (\mathit{Member}_{\alpha}\, r\, S)\, T\, v)\, u
\end{array}
$$

The reduction strategies for these operators are as follows: In all cases, the two terms to be compared are first reduced to normal form, and nothing further is done if this does not yield encodings of complex objects of the required type. Otherwise, the normal forms are substituted into the operator body. Then, for a term $(\mathit{Equal}_{\{\alpha\}}\, R\, S)$, 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 $(\mathit{Equal}_{\alpha_1 \times \cdots \times \alpha_k}\, t\, s)$, 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 $(\mathit{Member}_{\alpha}\, t\, R)$, reduction of the body involves a loop over the elements of R 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 $(\mathit{Subset}_{\{\alpha\}}\, R\, S)$ 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 relational operators of Chapter 3 and their reduction strategies work with no changes for complex objects, except that there are now more of each kind—one for each complex object type α instead of just one for each relation arity. We give the code for *Intersection* as an example.

$$
\mathit{Intersection}_{\{\alpha\}}{:}\, \{\alpha\} \to \{\alpha\} \to \{\alpha\} \equiv
$$

$$
\begin{array}{l}
\lambda R\colon\{\alpha\}.\,\lambda S\colon\{\alpha\}.\\
\quad \lambda c\colon \alpha \to \tau \to \tau.\,\lambda n\colon\tau.\\
\qquad R\,(\lambda r\colon\alpha.\,\lambda T\colon\tau.\,(\mathit{Member}_\alpha\, r\, S)\,(c\, r\, T)\,T)\,n
\end{array}
$$

The new operators *Tup*, *Tupcomb*, and *Set* can be implemented in a straightforward fashion:

$$
\begin{array}{l}
\mathit{Tup}\colon\{\alpha\} \to \{\alpha^1\} \equiv\\
\quad \lambda R\colon\{\alpha\}.\\
\qquad \lambda c\colon \alpha^1 \to \tau \to \tau.\,\lambda n\colon\tau.\\
\qquad\quad R\,(\lambda r\colon\alpha.\,\lambda T\colon\tau.\,c\,(\lambda f\colon\alpha \to \tau.\,f\,r)\,T)\,n
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{Set}\colon\{\alpha\} \to \{\{\alpha\}\} \equiv\\
\quad \lambda R\colon\{\alpha\}.\\
\qquad \lambda c\colon \{\alpha\} \to \tau \to \tau.\,\lambda n\colon\tau.\\
\qquad\quad R\,(\lambda r\colon\alpha.\,\lambda T\colon\tau.\,c\,(\lambda c'\colon\alpha \to \tau \to \tau.\,\lambda n'\colon\tau.\,c'\,r\,n')\,T)\,n
\end{array}
$$

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 of Chapter 3, 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.

$$
\begin{array}{l}
\mathit{Setcomb}\colon\{\{\alpha\}\} \to \{\alpha\} \equiv\\
\quad \lambda S\colon\{\{\alpha\}\}.
\end{array}
$$

$$
\begin{aligned}
&\lambda c{:}\, \alpha \to \tau \to \tau.\, \lambda n{:}\, \tau. \\
&\quad S\,(\lambda s{:}\, \{\alpha\}.\, \lambda T{:}\, \tau. \\
&\qquad s\,(\lambda x{:}\, \alpha.\, \lambda T'{:}\, \tau.\, (\mathit{First}_\alpha\; x\; s\; S)\,(c\; x\; T')\, T')\, T)\, n,
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{First}_\alpha{:}\, \alpha \to \{\alpha\} \to \{\{\alpha\}\} \to \mathsf{Bool} \equiv \\
&\quad \lambda x{:}\, \alpha.\, \lambda s{:}\, \{\alpha\}.\, \lambda S{:}\, \{\{\alpha\}\}. \\
&\qquad \lambda u{:}\, \tau.\, \lambda v{:}\, \tau. \\
&\qquad\quad S\,(\lambda s'{:}\, \{\alpha\}.\, \lambda T{:}\, \tau. \\
&\qquad\qquad s'\,(\lambda x'{:}\, \alpha.\, \lambda T'{:}\, \tau.\, \mathit{Equal}_\alpha\; x\; x'\,(\mathit{Equal}_{\{\alpha\}}\; s\; s'\; u\; v)\, T')\, T)\, u.
\end{aligned}
$$

5.3 Powerset

The one remaining operator, *Powerset*, which accounts for the great expressiveness of the algebra, poses a problem. Writing a λ-term that computes it is not too difficult—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:

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

Thus, to compute the powerset of a set S (represented as an iterator), we start out with the list $[\mathit{Nil}] \equiv \lambda c.\, \lambda n.\, c\,(\lambda c'.\, \lambda n'.\, n')\, n$ 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:

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

where

$$
\mathit{Prepend}{:}\, \alpha \to \{\{\alpha\}\} \to \{\{\alpha\}\} \equiv
$$

$$
\begin{aligned}
&\lambda x{:}\,\alpha.\,\lambda P{:}\,\{\{\alpha\}\}.\\
&\quad \lambda c{:}\,\{\alpha\} \rightarrow \tau \rightarrow \tau.\,\lambda n{:}\,\tau.\\
&\qquad P\,(\lambda s{:}\,\{\alpha\}.\,\lambda P'{:}\,\tau.\,c\,(\mathit{Cons}\,x\,s)\,P')\,n
\end{aligned}
$$

and

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

SimplePowerset is well-typed and computes the powerset function, but it is not good enough for use with the other algebra operators. The reason is that it introduces an "inhomogeneity" in either its input or output type in the following sense.

A type such as $\{\alpha\}$ has free occurrences of the type variable τ at many different levels. In principle, these could all correspond to different types (the most general typing of an encoding of a complex object would in fact assign different type variables to all these occurrences). However, structure-traversing operators like *Equal* or *Member* actually force all these occurrences to correspond to the same type. An expression $(\mathit{Equal}_{\{\alpha\}}\,R\,S)$ can be typed with any type β substituted for τ, as long as the substitution is carried out for *all* occurrences of τ. Now, the *SimplePowerset* operator forces the type of its input to be $(\alpha \rightarrow \{\{\alpha\}\} \rightarrow \{\{\alpha\}\}) \rightarrow \{\{\alpha\}\} \rightarrow \{\{\alpha\}\}$, i. e., $\{\alpha\}$ with $\{\{\alpha\}\}$ substituted for the *topmost* occurrences of τ only. Thus, once a relation R is used as input to $\mathit{SimplePowerset}_{\alpha}$, it cannot be used as input to any operator involving *Equal* or *Member*. We could get around this 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.

The solution to this problem is an extension of the "type-laundering" technique introduced in Section 4.2. 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 type homogeneous, so that they are usable with other operators. Thus, our goal is to construct a *Powerset* operator of type $\{\alpha\}\,[\tau{:=}\,\{\{\alpha\}\}] \rightarrow \{\{\alpha\}\}$, which takes a set of objects of type

$\alpha\,[\tau{:}=\{\{\alpha\}\}]$ and produces its powerset, but such that the objects occurring in the output have type just α. The following variant of *SimplePowerset* accomplishes this:

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

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, which we will define below, computes the identity function on complex objects of type α, but such that its input has type $\alpha\,[\tau{:}=\{\{\beta\}\}]$, whereas its output has type just α. Thus, the use of $(\mathit{Copy}_{\alpha,\{\{\alpha\}\}}\,x)$ instead of x does not change the semantics of the *Powerset* operator, but it assures that the output has the "clean" type $\{\{\alpha\}\}$ instead of $\{\{\alpha\,[\tau{:}=\{\{\alpha\}\}]\}\}$.

As in Section 4.2, 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.

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

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

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

It is straightforward to verify that this term 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:

$$
\begin{array}{l}
\mathit{Copy}_{\mathsf{o},\{\{\beta\}\}} \colon \mathsf{o}\,[\tau := \{\{\beta\}\}] \rightarrow \mathsf{o} \equiv \\
\quad \lambda x \colon \mathsf{o}.\, x
\end{array}
$$

$$
\begin{array}{l}
\mathit{Copy}_{\{\alpha\},\{\{\beta\}\}} \colon \{\alpha\}\,[\tau := \{\{\beta\}\}] \rightarrow \{\alpha\} \equiv \\
\quad \lambda S \colon \{\alpha\}\,[\tau := \{\{\beta\}\}]. \\
\qquad \lambda c \colon \alpha \rightarrow \tau \rightarrow \tau.\, \lambda n \colon \tau. \\
\qquad\quad S\ (\lambda s \colon \alpha\,[\tau := \{\{\beta\}\}].\, \lambda T \colon \{\{\beta\}\}. \\
\qquad\qquad \lambda c' \colon \{\beta\} \rightarrow \tau \rightarrow \tau.\, \lambda n' \colon \tau. \\
\qquad\qquad\quad c\,(\mathit{Copy}_{\alpha,\{\{\beta\}\}}\, s)\,(T\, c'\, n')) \\
\qquad\qquad (\lambda c' \colon \{\beta\} \rightarrow \tau \rightarrow \tau.\, \lambda n' \colon \tau.\, n) \\
\qquad\qquad (\lambda s' \colon \{\beta\}.\, \lambda S' \colon \tau.\, S')\, n
\end{array}
$$

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

A straightforward, but tedious induction shows that these terms compute the identity function on encodings of complex objects and can indeed be typed in the manner shown. Thus, the encoding of the *Powerset* operator is complete.

There is, however, one further issue. Although both input and output type 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 does not affect the use of S with other relational operators, e. g., $(\mathit{Set}\, S)$, because these do not make any assumptions about τ except that it corresponds to the same type everywhere, but it makes it impossible to use S (or any relation computed from S) in an expression $(\mathit{Powerset}_\beta\, S)$ where $\beta \neq \alpha$. Furthermore, S cannot be combined with the output of $(Powerset_\alpha\, S)$ in any way—for example, the expression $(Equal\,(Set\, S)\,(Powerset_\alpha\, S))$ would not type because $(Powerset_\alpha\, S)$ has type $\{\{\alpha\}\}$, whereas $(Set\, S)$ would be typed as $\{\{\alpha\}\}\,[\tau := \{\{\alpha\}\}]$ due to the "contaminated" type of S.

To get around this problem, we have to insert a type-laundering operator whenever a conflict occurs. For example, the expression $(Equal\,(Set\, S)\,(Powerset_\alpha\, S))$ can be made typable by writing it as $(Equal\,(Set\,(Copy_{\{\alpha\},\{\{\alpha\}\}}\, S)\,(Powerset_\alpha\, S))$.

To make this "laundering" process precise, let us fix a TLC$^=$ expression $\phi\,(R_1, \ldots, R_l)$ build from relational variables $R_1, \ldots, R_l$ and complex object algebra operators and study its typing in detail. We can think of ϕ as a circuit C_ϕ having input gates labeled $R_1, \ldots, R_l$ and interior gates labeled with algebra operators. Let $\nu_1, \ldots, \nu_n$ be an enumeration of the nodes of C_ϕ such that $\nu_1, \ldots, \nu_l$ are the input nodes and the number of each interior node is higher than that of any of its inputs. Let $\nu_{i_1}, \ldots, \nu_{i_m}$, where $1 < i_1 \leq \cdots \leq i_m \leq n$, be the nodes corresponding to *Powerset* operators and let $\{\{\alpha_1\}\}, \ldots, \{\{\alpha_m\}\}$ be the output types of these operators.

Now, we type ϕ as follows. Operators corresponding to nodes $\nu_{i_m}, \ldots, \nu_n$ are typed as usual, with no substitutions applied to τ. Operators $\nu_{i_{m-1}}, \ldots, \nu_{i_m - 1}$ are typed with the substitution $\tau := \{\{\alpha_m\}\}$ applied to their normal types, operators $\nu_{i_{m-2}}, \ldots, \nu_{i_{m-1}-1}$ are typed with the substitutions $\tau := \{\{\alpha_{m-1}\}\}$ and then $\tau := \{\{\alpha_m\}\}$ applied to their normal types, and so forth, until we end up with a chain of substitutions $[\tau := \{\{\alpha_m\}\}] \circ \cdots \circ [\tau := \{\{\alpha_1\}\}]$ applied to the types of $R_1, \ldots, R_l$.

Let us denote the nodes $\nu_{i_{j-1}+1}, \ldots, \nu_{i_j - 1}$ as *contamination layer* j, because their types are "contaminated" by the substitution $\gamma_j \equiv [\tau := \{\{\alpha_m\}\}] \circ \cdots \circ [\tau := \{\{\alpha_j\}\}]$. There is a typing conflict iff the output of a node ν from contamination layer j serves as input to a node ν' from contamination layer $k > j$. In this case, we pass the output S of ν through a "laundering pipeline"

$$(Copy_{\beta,\{\{\alpha_{k-1}\}\}}\,(Copy_{\beta,\{\{\alpha_{k-2}\}\}} \ldots (Copy_{\beta,\{\{\alpha_j\}\}}\, S) \ldots)),$$

where β is the type of S with no substitutions applied. Each stage $Copy_{\beta,\{\{\alpha_l\}\}}$ of this "pipeline" imposes a substitution $\tau := \{\{\alpha_l\}\}$ on the type of its input, without changing its value. Thus, if the output of the "pipeline" is typed as $\gamma_k\,(\beta)$, as required by node ν', then the input gets type $(\gamma_k \circ [\tau := \{\{\alpha_{k-1}\}\}] \circ \cdots \circ [\tau := \{\{\alpha_j\}\}])\,(\beta) \equiv \gamma_j\,(\beta)$, which matches the output type of node ν.

Suppose, for example, we want to type $(Equal\,(Set\, S)\,(Powerset_\alpha\, S))$ according to this algorithm. Let us order the operators in this expression as: S, $Powerset_\alpha$, *Set*, *Equal*. Then there are two contamination layers, divided by the $Powerset_\alpha$ operator: a "clean" one containing the *Set* and *Equal* operators, with no substitutions applied to their types, and a "dirty" one containing S, with the substitution $\tau := \{\{\alpha\}\}$ applied to its type. There is one typing conflict because node S from the "dirty" layer serves as input to node *Set* from the "clean" layer. Thus, we have to insert a (one-stage) laundering pipeline $\mathit{Copy}_{\{\alpha\},\{\{\alpha\}\}}$, which leads to the typable expression $(\mathit{Equal}\,(\mathit{Set}\,(\mathit{Copy}_{\{\alpha\},\{\{\alpha\}\}}\,S)\,(\mathit{Powerset}_\alpha\,S))$ already mentioned above.

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 contamination level l must be typed as $\mathsf{o} \to \mathsf{o} \to \gamma_l(\tau) \to \gamma_l(\tau) \to \gamma_l(\tau)$ instead of $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$. 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 2.5), we can do this by replacing every occurrence of *Eq* at contamination level l with the term

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

where the types $\beta_1, \ldots, \beta_k$ are given by $\gamma_l(\tau) = \beta_1 \to \cdots \to \beta_k \to \tau$. For every pair of constants o_i, o_j, the normal form of $(\mathit{Eq}\, o_i\, o_j)$ arises from the normal form of $(\mathit{Eq}_l\, o_i\, o_j)$ by a series of η-reductions [6]. It is known that in a $\beta\eta$-reduction sequence, η-reductions can always be pushed to the end [6, Theorem 15.1.6]. This remains true even if *Eq*-reductions are added, because *Eq*- and η-reductions commute for 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 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.

Putting everything together, we arrive at the following theorem:

Theorem 5.3.1 *Let $\phi(R_1, \ldots, R_l)$ be a complex object algebra expression over the relational schema $\mathcal{S} = (R_1, \ldots, R_l)$. Then there exists a term ψ of the simply typed λ-calculus with equality such that for every instance $(r_1, \ldots, r_l)$ of $\mathcal{S}$, the expression $(\psi\, \overline{r_1} \ldots \overline{r_l})$ reduces to $\overline{\phi(r_1, \ldots, r_l)}$, where $\overline{r}$ denotes the encoding of relation r. Moreover, if the Powerset operator* *is not used, the normal form of* $(\psi\,\overline{r_1}\ldots\overline{r_l})$ *can be computed in time polynomial in the size of* $r_1,\ldots,r_l$.

Chapter 6

Query Terms of Fixed Order

In the previous chapters, we have been concerned with the feasibility of embedding database queries in the simply typed λ-calculus at all. Now that we know that queries up to elementary time are representable, a natural question is whether it is possible to obtain a connection between the space/time complexity of queries and the structure of the λ-terms that represent them. A positive answer to this question would on the one hand enable us to define functional query languages that express just the feasible queries, and on the other hand give us new insights about the expressive power of various fragments of TLC.

In this and the next chapter, we show that there is indeed such a connection, and in fact a very simple one. Under suitable input/output conventions, we obtain an exact correspondence between the complexity classes FIRST-ORDER, PTIME, PSPACE, and EXPTIME and fixed-order fragments of $\mathrm{TLC}^=$. The proof of this correspondence involves three parts: specifying the exact input/output conventions and the query format used, obtaining lower bounds on the expressibility of fixed-order fragments by simulating generic computations, and obtaining matching upper bounds by defining efficient evaluation algorithms for these fragments. In the current chapter, we define the input/output framework and the languages we will use, and we give a series of encodings that establish lower bounds on the expressive power of these languages. In the next chapter, we give an analysis of the structure of query terms and show how to evaluate them efficiently.

6.1 Minimizing the Input/Output Order

In the encoding scheme used so far, a k-ary relation is encoded as a term

$$\lambda c{:}((\mathsf{o} \to \cdots \to \mathsf{o} \to \tau) \to \tau) \to \tau \to \tau.\, \lambda n{:}\tau.$$

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

whose type $\{\mathsf{o}^k\}$ is of order 4. The relational operators of Chapter 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 Chapter 4 are of even higher order because of "output 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 of Section 2.4 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 $O = \{o_1, o_2, \ldots\}$ be the set of constants. For convenience, we assume that this set of constants also serves as the universe over which relations are defined. If $r = \{(o_{1,1}, o_{1,2}, \ldots, o_{1,k}), \ldots, (o_{m,1}, o_{m,2}, \ldots, o_{m,k})\} \subseteq O^k$ is a k-ary relation over O, then the encoding $\overline{r}$ of r is the λ-term

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

The difference between this encoding scheme and the one of Section 2.4 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 $(\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau) \to$

$\tau \to \tau$, where τ is some fixed type variable.[1] The order of this type is 2, independent of the arity of r. We abbreviate this type as o_k^τ. Instances of this type, obtained by substituting some type expression θ for τ, are abbreviated as o_k^θ, or, if the exact nature of θ does not matter, as o_k^*.

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

Lemma 6.1.1 *Let f be any TLC$^=$ term without free variables and in normal form (i. e., with no beta- or Eq-reduction possible) of type o_k^τ, where τ is a type variable different from o. Then either $f \equiv \lambda c.\, c\, o_{1,1} \ldots o_{1,k}$ or $f \equiv \overline{r}$ for some relation $r \subseteq O^k$.*

Proof: Clearly, f cannot be build from Eq and constants alone. Thus, f must be of the form $\lambda c{:}\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, g$, where g is a term of type $\tau \to \tau$. If g does not begin with a λ-abstraction, then g must be of the form $c\, o_{1,1} \ldots o_{1,k}$. Note that Eq cannot occur in g, because the only possible subterms of g of type o are explicit constants, and an occurrence of Eq with constant arguments would create a redex. If g does begin with a λ-abstraction, then $g \equiv \lambda n{:}\tau.\, h$, where h is of type τ. Again, Eq cannot occur in h, and it is easy to see that the most general term of type τ that can be build from constants, c, and n is a list $c\, \vec{o}_1\, (c\, \vec{o}_2 \ldots (c\, \vec{o}_m\, n) \ldots)$, where $m \geq 0$. Hence, in this case $f \equiv \overline{r}$ for some relation $r \subseteq O^k$.$\Box$

Remark: The two terms $\lambda c.\, c\, o_{1,1} \ldots o_{1,k}$ and $\lambda c.\, \lambda n.\, c\, o_{1,1} \ldots o_{1,k}\, n$, η-convert to each other and they cannot be distinguished at the type level. For this reason, we extend our encoding convention to allow both forms as valid representations of relations containing just one tuple.

6.2 Language Definitions

We now define the query languages that we will consider. We define two sets of languages TLI$_i^=$ and MLI$_i^=$, where $i = 0, 1, 2, \ldots$, as fixed-order fragments of TLC$^=$ and core-ML$^=$, respectively. Membership in these languages is defined by a typing constraint that guarantees that terms in these languages behave as queries, i. e., when given a set of encodings of relations as input, they produce the encoding of another relation as output. For purposes of comparison we use the same syntax in both TLI and MLI definitions. That is, in MLI we interpret the outermost λ's as `let`'s. The other `let`'s in MLI can be eliminated, if desired, by replacing every subterm of the form "`let` $x = N$ `in` M" with $M\,[x{:=}\,N]$.

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

Definition 6.2.1 *A query program of arity* $(k_1, \ldots, k_l, k)$ *in* $\mathrm{TLI}_i^=$ *(the language of* typed list iteration of order i with equality*) is a typed TLC*$^=$ *term* Q *of order* $i+3$ *such that:* Q *has the form* $\lambda R_1 \ldots \lambda R_l.\, M$ *and for every database of arity* $(k_1, \ldots, k_l)$ *encoded by* $\overline{r_1} \ldots \overline{r_l}$ *it is possible to type* $(\lambda R_1 \ldots \lambda R_l.\, M)\, \overline{r_1} \ldots \overline{r_l}$ *as* o_k^τ.

Definition 6.2.2 *A query program of arity* $(k_1, \ldots, k_l, k)$ *in* $\mathrm{MLI}_i^=$ *(the language of* ML-typed list iteration of order i with equality*) is a typed core-ML*$^=$ *term* Q *of order* $i+3$ *such that:* Q *has the form* $\lambda R_1 \ldots \lambda R_l.\, M$ *and for every database of arity* $(k_1, \ldots, k_l)$ *encoded by* $\overline{r_1} \ldots \overline{r_l}$ *it is possible to ML-type* $(\lambda R_1 \ldots \lambda R_l.\, M)\, \overline{r_1} \ldots \overline{r_l}$ *as* o_k^τ *with the bindings* $\lambda R_1 \ldots \lambda R_l$ *typed as* `let`*'s.*

We use order $i + 3$ in these definitions because the expressive power of list iteration really depends on the order of the objects that are manipulated during an iteration, not on the order of the term in which the iterator is used. An iterator over objects of type σ of order i has type $(\mathsf{o} \to \cdots \to \mathsf{o} \to \sigma \to \sigma) \to \sigma \to \sigma$, thus its order is $i + 2$. If a query term abstracts on such an iterator, the order of the query term becomes $i + 3$, but the relevant measure is still i.

The above definitions are semantic because they involve quantification over all inputs. By Lemma 6.1.1 and the fact that τ is a type variable different from o, it is easy to see that every program in these languages is guaranteed to have a correct output given correct inputs. These semantic definitions can be made syntactic:

Lemma 6.2.3 *Given* $(k_1, \ldots, k_l, k)$ *and a typed* λ*-term* $(\lambda R_1 \ldots \lambda R_l.\, M)$ *of TLC*$^=$ *or core-ML*$^=$ *of order* $i+3$ *one can efficiently decide if it is a query program of arity* $(k_1, \ldots, k_l, k)$ *of TLI*$_i^=$ *or MLI*$_i^=$*. Moreover, all inputs to this term can be typed with the same monomorphic type.*

Proof: As stated in Section 6.1, the principal type of an encoding of a relation is $(\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau) \to \tau \to \tau$, independent of the content or the size of the relation (as long as there are at least two tuples). Thus, in order to decide whether a term Q is a $\mathrm{TLI}_i^=$ or $\mathrm{MLI}_i^=$ query program of arity $(k_1, \ldots, k_l, k)$, it suffices to type-check an expression $(Q\, \overline{r_1} \ldots \overline{r_l})$, where $\overline{r_1}, \ldots, \overline{r_l}$ are encodings of relations of arities $k_1, \ldots, k_l$ and length 2. □

In the setting of these query languages input and output terms are monomorphically typed. Unlike [19, 43], we allow that the monomorphic types of inputs and outputs differ. Outputs are always typed as o_k^τ but inputs can be typed with some type expression θ substituted for τ, where the order of θ is bounded by the index of the language $\mathrm{TLI}_i^=$ or $\mathrm{MLI}_i^=$ the query term is in.

6.3 Embedding Database Queries in $\mathrm{TLI}_i^=$ and $\mathrm{MLI}_i^=$

In this section, we prover lower bounds on the expressive power of the $\mathrm{TLI}_i^=$ and $\mathrm{MLI}_i^=$ languages by encoding various classes of queries. Our encodings show that $\mathrm{TLI}_0^=$ expresses relational algebra and order and that $\mathrm{TLI}_1^=$ expresses all PTIME queries. If, in addition to the typing $Eq\colon \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ prescribed in Section 2.1, we also allow Eq to be typed as $Eq\colon \mathsf{o} \to \mathsf{o} \to \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ (thereby introducing a weak form of polymorphism), we obtain a version $\mathrm{TLI}_0^{\approx}$ of $\mathrm{TLI}_0^=$ that expresses relational algebra, parity, majority, and (deterministic) graph accessibility. For higher orders, we use a generic simulation to show that $\mathrm{TLI}_{2k+1}^=$ and $\mathrm{MLI}_{2k+1}^=$ express k-EXPTIME and $\mathrm{TLI}_{2k+2}^=$ and $\mathrm{MLI}_{2k+2}^=$ express k-EXPSPACE. Here, k-EXPTIME stands for time complexity $2^{2^{\cdot^{\cdot^{2^{n^{\mathcal{O}(1)}}}}}}$, where there is a tower of k 2's, and k-EXPSPACE is defined analogously.

6.3.1 Embeddings in $\mathrm{TLI}_0^=$

The encodings of the relational operators can be adopted to our new input/output convention in a straightforward fashion. We illustrate this for the *Equal*, *Member*, and *Union* operators only. The basic idea is that whenever an operator was passed a k-tuple argument, it is now passed the k individual components of the tuple instead.

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

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

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

With these modifications, all relational operators can be expressed as $\mathrm{TLI}_0^=$ terms. However, $\mathrm{TLI}_0^=$ expresses more than just relational algebra: it is also possible to define an *order* relation on the input universe, using the order inherent in the encoding of the input. Namely, if $(\overline{r_1}, \ldots, \overline{r_l})$ is an encoded database of arities $(k_1, \ldots, k_l)$, we can compute the input universe as described in Section 4.1:

$$
\begin{aligned}
&D{:}\,\mathsf{o}_1^{\tau} \equiv \\
&\quad (\mathit{Union}_1\,(\mathit{Project}_{k_1;1}\,\overline{r_1})\,(\mathit{Union}_1\,(\mathit{Project}_{k_1;2}\,\overline{r_1})\ldots \\
&\qquad (\mathit{Union}_1\,(\mathit{Project}_{k_2;1}\,\overline{r_2})\,(\mathit{Union}_1\,(\mathit{Project}_{k_2;2}\,\overline{r_2})\ldots \\
&\qquad\quad \ldots \\
&\qquad\qquad (\mathit{Union}_1\,(\mathit{Project}_{k_l;1}\,\overline{r_l})\,(\mathit{Union}_1\,(\mathit{Project}_{k_l;2}\,\overline{r_l})\ldots),
\end{aligned}
$$

and then check whether x precedes y in D by means of the following λ-term:

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

This term works by looping over the constants in D until either x or y is seen. If x is seen first, it reduces to *True*, otherwise it reduces to *False*.

From the above and the equivalence of relational algebra and calculus, it follows that:

Theorem 6.3.1 *Under the input/output conventions of Section 6.1, all first-order queries over ordered structures can be expressed in* $\mathit{TLI}_0^=$ *and* $\mathit{MLI}_0^=$*.*

6.3.2 Embeddings in $\mathbf{TLI}_0^{\simeq}$

$\mathrm{TLI}_0^{\simeq}$ allows us to type *Eq* polymorphically either as $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ or $\mathsf{o} \to \mathsf{o} \to \mathsf{o} \to \mathsf{o} \to \mathsf{o}$. The added freedom makes it possible to express iterations that maintain *state*, by passing a constant from one stage to the next and basing the action taken at each stage on the value of the constant that is passed in. With this technique, we can express non-relational queries such as *Parity* and *Majority*.

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

$$
\begin{array}{l}
Parity\colon \mathsf{o}_k^{\mathsf{o}} \to \mathsf{o}_1^{\tau} \equiv \\
\quad \lambda R\colon \mathsf{o}_k^{\mathsf{o}}. \\
\quad\quad \lambda c\colon \mathsf{o} \to \tau \to \tau.\, \lambda n\colon \tau. \\
\quad\quad\quad c\,(R\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}.\, \lambda P\colon \mathsf{o}.\, (Eq\, P\, 0)\, 1\, 0)\, 0)\, n
\end{array}
$$

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

$$
\begin{array}{l}
Majority\colon \mathsf{o}_2^{\mathsf{o}} \to \mathsf{o}_1^{\tau} \equiv \\
\quad \lambda R\colon \mathsf{o}_2^{\mathsf{o}}. \\
\quad\quad \lambda c\colon \mathsf{o} \to \tau \to \tau.\, \lambda n\colon \tau. \\
\quad\quad\quad c\,(Compare_R \\
\quad\quad\quad\quad (R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, (Eq\, x_2\, 1)\,(Succ_R\, T)\, T)\, First_R) \\
\quad\quad\quad\quad (R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, (Eq\, x_2\, 0)\,(Succ_R\, T)\, T)\, First_R) \\
\quad\quad\quad\quad 0\, 0\, 1)\, n
\end{array}
$$

Here, $First_R$, $Succ_R$, and $Compare_R$ are functions that operate on the "labels" in the first column of R. $First_R$ returns the label of the first tuple in R, $(Succ_R\, x)$ returns the label of the tuple following the one labeled x, and $(Compare_R\, x\, y\, a\, b\, c)$ compares the positions in R of the tuples labeled x and y, reducing to a if x precedes y, to b if x and y are equal, and to c if y precedes x. These terms can be written as follows:

$$First_R \equiv R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, x_1)\, 0$$

$$(Succ_R\, x) \equiv R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, Compare_R\, x_1\, x\, T\, T\, x_1)\, x$$

$$(Compare_R\, x\, y\, a\, b\, c) \equiv R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, Eq\, x\, y\, b\,(Eq\, x_1\, x\, a\,(Eq\, x_1\, y\, c\, T)))\, b$$

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

$$
\begin{aligned}
&\mathit{DGAP}\colon \mathsf{o}_2^{\mathsf{o}} \to \mathsf{o}_2^{\mathsf{o}} \to \mathsf{o}_1^{\tau} \equiv \\
&\quad \lambda R\colon \mathsf{o}_2^{\mathsf{o}}.\, \lambda S\colon \mathsf{o}_2^{\mathsf{o}}. \\
&\qquad \lambda c\colon \mathsf{o} \to \tau \to \tau.\, \lambda n\colon \tau. \\
&\qquad\quad c\,(S\,(\lambda uv\colon \mathsf{o}.\, \lambda W\colon \mathsf{o}.\, \mathit{Eq}\, v\, (\mathit{Ancestor}\, u)\, 1\, 0)\, 0)\, n,
\end{aligned}
$$

where

$$(\mathit{Ancestor}\, u) \equiv R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, (\mathit{Eq}\, T\, v)\, T\, (\mathit{Parent}\, T))\, u$$

and

$$(\mathit{Parent}\, v) \equiv R\,(\lambda x_1\colon \mathsf{o}.\, \lambda x_2\colon \mathsf{o}.\, \lambda T\colon \mathsf{o}.\, (\mathit{Eq}\, x_1\, v)\, x_2\, T)\, v$$

It is interesting to note that deterministic graph accessibility is LOGSPACE-complete for first-order reductions [28], but only if vertices can be labeled by *tuples of constants.* This means that an instance of the problem consists of a $2\,k$-ary relation R such that each tuple $(x_1, \ldots, x_k, y_1, \ldots, y_k) \in R$ denotes an edge from the vertex labeled $(x_1, \ldots, x_k)$ to the vertex labeled $(y_1, \ldots, y_k)$.

It seems that this more general version of graph accessibility cannot be expressed in $\mathrm{TLI}_0^{\simeq}$, since it requires list iteration over *tuples* of constants, which cannot be encoded as $\mathrm{TLC}^{=}$ objects of order 0. Thus, the expressive power of $\mathrm{TLI}_0^{\simeq}$ appears to fall short of LOGSPACE. It is possible to express all of LOGSPACE by adding tuples of constants as primitive objects to the language, but this would sacrifice the simplicity of the framework to some extent. (Using the evaluation techniques presented in the next chapter, it can be shown that $\mathrm{TLI}_0^{\simeq}$ queries, even with an additional tupling constructor, can be evaluated in LOGSPACE; so $\mathrm{TLI}_0^{\simeq}$ + tuples = LOGSPACE.)

6.3.3 Embeddings in $\mathrm{TLI}_1^{=}$ and $\mathrm{MLI}_1^{=}$

The modifications from Section 4.1 for encoding fixpoint queries are not so straightforward. The problem is that even with order 2 representations of relations, the order of a fixpoint query is

$$\text{order}\,((\mathsf{o}^\tau_{k_1} \to \cdots \to \mathsf{o}^\tau_{k_l})\,[\tau := \mathsf{o}^\tau_k] \to \mathsf{o}^\tau_k) = 5.$$

To bring the order down to 4, we use the following trick: 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

$$f_r : \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \mathsf{Bool}$$

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

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

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

If r is given as an iterator $R : \mathsf{o}^\tau_k$, then a characteristic function representation of r is computed by the TLC$^=$ term

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

In order to translate back from a characteristic function to an iterator, we need a list $D : \mathsf{o}^\tau_1$ containing all the constants in the 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 $f_r : \chi_k$ is computed by the TLC$^=$ term

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

Suppose now that the TLC$^=$ term $(\lambda R.\, Q)\colon \mathsf{o}^\tau_k \to \mathsf{o}^\tau_k$ 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, as in Section 4.1,

$$\mathit{Fix}_Q \equiv \lambda R_1 \ldots \lambda R_l.\, \mathit{Crank}\,(\lambda f.\, Q')\,(\mathit{ListToChar}_k\, \mathit{Nil}),$$

where *Crank* is a suitably large Church numeral derived from a cross product of the input relations and *Nil* is the empty list. However, this term can be typed only with either ML-polymorphism or a certain amount of type laundering. More precisely, the occurrences of R_i in *Crank* have to have a "contaminated" type $(\mathsf{o}^\tau_{k_i})\,[\tau{:}=\chi_k]$, whereas the occurrences of R_i in Q' have to have a "clean" type $\mathsf{o}^\tau_{k_i}$, for $1 \le i \le l$. This can be achieved by either typing R_i polymorphically or by replacing every occurrence of R_i in Q' by the term $(\mathit{Copy}_{k_i}\, R_i)$, where Copy_{k_i} is the following "laundering" operator:

$$
\begin{aligned}
&\mathit{Copy}_{k_i}\colon (\mathsf{o}^\tau_{k_i})\,[\tau{:}=\chi_k] \to \mathsf{o}^\tau_{k_i} \equiv\\
&\quad \lambda R\colon (\mathsf{o}^\tau_{k_i})\,[\tau{:}=\chi_k].\\
&\qquad \lambda c\colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau.\, \lambda n\colon \tau.\\
&\qquad\quad (R\ (\lambda x_1\colon \mathsf{o} \ldots \lambda x_{k_i}\colon \mathsf{o}.\, \lambda T\colon \chi_k.\\
&\qquad\qquad\qquad \lambda y_1\colon \mathsf{o} \ldots \lambda y_k\colon \mathsf{o}.\, \lambda u\colon \tau.\, \lambda v\colon \tau.\, c\, x_1 \ldots x_{k_i}\, (T\, y_1 \ldots y_k\, u\, v))\\
&\qquad\qquad (\lambda y_1\colon \mathsf{o} \ldots \lambda y_k\colon \mathsf{o}.\, \lambda u\colon \tau.\, \lambda v\colon \tau.\, n))\, z_1 \ldots z_k\, n\, n
\end{aligned}
$$

(Here, $z_1, \ldots, z_k$ are some variables of type o.)

Thus, the final encoding of the fixpoint query in TLI$^=_1$ becomes

$$
\begin{aligned}
&\mathit{Fix}_Q\colon (\mathsf{o}^\tau_{k_1} \to \cdots \to \mathsf{o}^\tau_{k_l})\,[\tau{:}=\chi_k] \to \mathsf{o}^\tau_k \equiv\\
&\quad \lambda R_1\colon \mathsf{o}^\tau_{k_1}\,[\tau{:}=\{\mathsf{o}^k\}] \ldots \lambda R_l\colon \mathsf{o}^\tau_{k_l}\,[\tau{:}=\{\mathsf{o}^k\}].\\
&\qquad \mathit{Crank}\\
&\qquad\quad (\lambda f\colon \chi_k.\, Q'\,[R_1{:}=(\mathit{Copy}_{k_1}\, R_1), \ldots, R_l{:}=(\mathit{Copy}_{k_l}\, R_l)])\\
&\qquad\quad (\mathit{ListToChar}_k\, \mathit{Nil}).
\end{aligned}
$$

It is easy to see that this term is indeed well typed and that its type is of order 4. Hence, we have

Theorem 6.3.2 *Under the input/output conventions of Section 6.1, all PTIME queries can be expressed in* $TLI_1^=$ *and* $MLI_1^=$*.*

Note that the theorem 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 is the subject of the next chapter.

6.3.4 Embeddings in $\mathbf{TLI}_2^=$ and $\mathbf{MLI}_2^=$

The increase in order by 1 in going from $\mathrm{TLI}_1^=$ to $\mathrm{TLI}_2^=$ allows us to iterate first-order queries an *exponential* number of times. This is sufficient to evaluate any query Q of data complexity PSPACE, by just iterating a first-order representation of the transition function of a Turing machine computing Q [48].

To obtain an exponential iterator, we proceed as follows: We take the fixpoint query of the previous section,

$$\mathit{Fix}_Q \equiv \lambda R_1 \ldots \lambda R_l.\, \mathit{Crank}\,(\lambda f.\, Q)\,(\mathit{ListToChar}_k\, \mathit{Nil}),$$

where $\lambda f.\, Q$ is a first-order query of type $\chi_k \to \chi_k$, and replace the Church numeral *Crank* by its exponential

$$\mathit{Crank}' \colon \mathsf{Int} \equiv \mathit{Crank}\,(\lambda s.\, \lambda z.\, s\,(s\, z))$$

If *Crank* is the Church numeral n^k, where n is the database size, then Crank' is the Church numeral 2^{n^k}. The type of *Crank* in the term above is $\mathsf{Int}\,[\tau{:}= \tau \to \tau]$, i. e., $((\tau \to \tau) \to (\tau \to \tau)) \to (\tau \to \tau) \to (\tau \to \tau)$, which is of order 3. If we want to use Crank' to iterate the query $\lambda f.\, Q$ of type $\chi_k \to \chi_k$, then we have to type *Crank* as $\mathsf{Int}\,[\tau{:}= \chi_k \to \chi_k]$, which means that we have to type the input relations with τ substituted by $\chi_k \to \chi_k$. Since the order of $\chi_k \to \chi_k$ is 2, we obtain a $\mathrm{TLI}_2^=$ query term. Of course, the occurrences of $R_1, \ldots, R_l$ inside Q need a "clean" type; this is achieved by either using ML-typing or type laundering. Thus, we obtain:

Theorem 6.3.3 *Under the input/output conventions of Section 6.1, all PSPACE queries can be expressed in* $TLI_2^=$ *and* $MLI_2^=$*.*

6.3.5 Embeddings in $\mathbf{TLI}_3^=$ and $\mathbf{MLI}_3^=$

With a further increase in order, we can afford another exponential. Clearly, there is no point in iterating a first-order query a doubly exponential number of times over polynomially sized relations (of which there are only exponentially many)—the computation would loop or reach a fixpoint after at most a singly exponential number of iterations. Instead, we have to use the order increase to build exponentially-sized data structures.

The main idea is to build a domain of exponential size by using relations over the input universe as *atomic elements* of a larger domain. There are 2^{n^k} relations of arity k over a universe D of size n, so by using these relations as atomic objects, we obtain a domain D' of size 2^{n^k}. D' can be economically represented using a list of characteristic functions, which requires an order 3 type. We build D' using a modified version of the *Powerset* operator of Section 5.3 that uses characteristic functions:

$$\begin{aligned} D'\colon \{\chi_k\} \equiv D^k\ &(\lambda x_1{:}\mathsf{o} \ldots \lambda x_k{:}\mathsf{o}.\, \lambda S{:}\{\chi_k\}.\, \mathit{Union}\, S\, (\mathit{Prepend}\, x_1 \ldots x_k\, S)) \\ &(\lambda c{:}\chi_k \to \tau \to \tau.\, \lambda n{:}\tau.\, c\, (\mathit{ListToChar}_k\, \mathit{Nil})\, n), \end{aligned}$$

where

$$D^k \equiv (\overbrace{\mathit{Times}\, D\, (\mathit{Times}\, D \ldots (\mathit{Times}\, D\, D)}^{k \text{ factors}} \ldots))$$

and

$$\begin{aligned} &\mathit{Prepend}\colon \mathsf{o} \to \cdots \to \mathsf{o} \to \{\chi_k\} \equiv \\ &\quad \lambda x_1{:}\mathsf{o} \ldots \lambda x_k{:}\mathsf{o}.\, \lambda S{:}\{\chi_k\}. \\ &\qquad \lambda c{:}\chi_k \to \tau \to \tau.\, \lambda n{:}\tau. \\ &\qquad\quad S\, (\lambda f{:}\chi_k.\, \lambda T{:}\tau. \\ &\qquad\qquad c\, (\lambda y_1{:}\mathsf{o} \ldots \lambda y_k{:}\mathsf{o}.\, \lambda u{:}\tau.\, \lambda v{:}\tau. \\ &\qquad\qquad\quad (\mathit{Equal}_k\, x_1 \ldots x_k\, y_1 \ldots y_k)\, u\, (f\, y_1 \ldots y_k))\, T)\, n. \end{aligned}$$

This term yields D' as a list of the 2^{n^k} possible characteristic functions of arity k.

Equality and order predicates over the domain D' can be computed using pointwise comparison of characteristic functions:

$$\begin{aligned} &\mathit{Equal}_{\chi_k}\colon \chi_k \to \chi_k \to \mathsf{Bool} \equiv \\ &\quad \lambda f{:}\chi_k.\, \lambda g{:}\chi_k. \\ &\qquad \lambda u{:}\tau.\, \lambda v{:}\tau. \\ &\qquad\quad D^k\, (\lambda x_1{:}\mathsf{o} \ldots \lambda x_k{:}\mathsf{o}.\, \lambda T{:}\tau.\, \mathit{Xor}\, (f\, x_1 \ldots x_k)\, (g\, x_1 \ldots x_k)\, v\, T)\, u \end{aligned}$$

$$
\begin{aligned}
&\mathit{Precedes}_{\chi_k}\colon \chi_k \to \chi_k \to \mathsf{Bool} \equiv\\
&\quad \lambda f\colon \chi_k.\, \lambda g\colon \chi_k.\\
&\qquad \lambda u\colon \tau.\, \lambda v\colon \tau.\\
&\qquad\quad D^k\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}.\, \lambda T\colon \tau.\, \mathit{Xor}\,(f\,x_1 \ldots x_k)\,(g\,x_1 \ldots x_k)\,(g\,x_1 \ldots x_k)\,T)\,v
\end{aligned}
$$

Finally, we can build terms $\mathit{ListToChar}'$ and $\mathit{CharToList}'$ very much like those in Section 6.3.3 that use D' to convert back and forth between lists of characteristic functions and characteristic functions of characteristic functions. This allows us to (a) represent arbitrary relations over D' as order 2 objects, namely characteristic functions of characteristic functions, and (b) write arbitrary first-order queries that map relations over D' to relations over D'.

Suppose now that $\lambda f.\, Q$ is a first-order query over domain D', of type $\chi'_k \to \chi'_k$, where

$$
\chi'_k \equiv \overbrace{\chi_k \to \cdots \to \chi_k}^{k} \to \mathsf{Bool}
$$

is the type of a k-ary characteristic function over D'. We can iterate $\lambda f.\, Q$ an *exponential* number of times over an *exponential* domain by writing

$$
\mathit{Fix}_Q \equiv \mathit{Crank}'\,(\lambda f.\, Q)\,(\lambda g_1\colon \chi_k \ldots \lambda g_k\colon \chi_k.\, \mathit{False})
$$

where Crank' is the Church numeral 2^{n^k} obtained by exponentiating the length of D^k. As usual, type laundering or ML-polymorphism is needed to make this term typable. The order of Fix_Q is one more than the fixpoint query of the last section, because the "one-step" function $\lambda f.\, Q$ now has type $\chi'_k \to \chi'_k$ instead of $\chi_k \to \chi_k$; thus Fix_Q is a $\mathrm{TLI}_3^=$ query.

It is easy to see that terms of the form Fix_Q can capture any EXPTIME query, since we can just code up the transition function of the Turing machine computing the query as a first-order expression, which we then iterate over an exponential amount of tape encoded as a relation over D'. Thus, we have

Theorem 6.3.4 *Under the input/output conventions of Section 6.1, all EXPTIME queries can be expressed in $TLI_3^=$ and $MLI_3^=$.*

It is now easy to see how this process continues for higher orders. In going to $\mathrm{TLI}_4^=$, we use the additional order to code a *doubly exponential* iteration over an *exponential* universe, thus capturing EXPSPACE, then in $\mathrm{TLI}_5^=$ we can code a *doubly exponential* iteration over a universe of *doubly exponential* size, capturing 2-EXPTIME, and so forth. In general, we have the following theorem:

Theorem 6.3.5 *Under the I/O conventions of Section 6.1, all k-EXPTIME queries can be expressed in* $TLI^{=}_{2k+1}$ *and* $MLI^{=}_{2k+1}$*, and all k-EXPSPACE queries can be expressed in* $TLI^{=}_{2k+2}$ *and* $MLI^{=}_{2k+2}$*. Here, k-EXPTIME and k-EXPSPACE denote time and space complexity* $2^{2^{\cdot^{\cdot^{2^{n^{O(1)}}}}}}$*, where there is a tower of k 2's.*

Chapter 7

Evaluating $\mathrm{TLI}_i^=$ Queries

In this chapter, we give upper bounds on the data complexity of queries expressible in $\mathrm{TLI}_i^=$ and $\mathrm{MLI}_i^=$. In particular, we show that over ordered structures, $\mathrm{TLI}_0^=$ and $\mathrm{MLI}_0^=$ express just the first-order queries, $\mathrm{TLI}_1^=$ and $\mathrm{MLI}_1^=$ express just the PTIME queries, and that further increases in order lead to data complexity PSPACE and EXPTIME, respectively, and probably in general to k-EXPTIME and k-EXPSPACE, where $k = \lfloor (i-1)/2 \rfloor$. The proofs consist of specifying a suitable reduction strategy for query terms together with the design of space-efficient representations of intermediate results.

7.1 The Structure of $\mathrm{TLI}_i^=$ and $\mathrm{MLI}_i^=$ Terms

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

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

Since Q is in $\mathrm{TLI}_i^=$ or $\mathrm{MLI}_i^=$, it can be typed as $\mathsf{o}^*_{k_1} \to \cdots \to \mathsf{o}^*_{k_l} \to \mathsf{o}^\tau_k$, where the asterisks stand for unspecified types of order $\leq i$ built from o and τ (in the case of $\mathrm{MLI}_i^=$ terms, these types are to be interpreted polymorphically). We call any such typing a *canonical typing* of Q. Under a canonical typing, every subterm t of Q is assigned a certain type, which we call the *canonical type* of t for this canonical typing of Q.

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

Definition 7.1.1 *Let Q be a $TLI_i^=$ or $MLI_i^=$ query term and γ be a canonical typing of Q. We say that Q is in γ-*canonical form *if Q is in closed normal form and every complete subterm t of Q is of the form $\lambda x_1{:}\,\alpha_1 \ldots \lambda x_k{:}\,\alpha_k.\, M$, where $k \geq 0$ and $\alpha_1, \ldots, \alpha_k$ are such that the canonical type of t under γ is $\alpha_1 \to \cdots \to \alpha_k \to \sigma$, where σ is either τ or o. We say that Q is in canonical form if it is in γ-canonical form for some canonical typing γ.*

Lemma 7.1.2 *Let P be a $TLI_i^=$ or $MLI_i^=$ term mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then there is a $TLI_i^=$ or $MLI_i^=$ term Q in canonical form such that P and Q define the same database query, i. e., for every input $\overline{r_1}, \ldots, \overline{r_l}$, the normal forms of $(P\,\overline{R_1}, \ldots, \overline{R_l})$ and $(Q\,\overline{R_1}, \ldots, \overline{R_l})$ encode the same relation. Q can be effectively determined from P.*

Proof: Fix some canonical typing γ of P. We obtain Q from P by a series of η-expansions, where a complete subterm $\lambda x_1{:}\,\alpha_1 \ldots \lambda x_k{:}\,\alpha_k.\, M$ with $\mathrm{type}(M) = \alpha_{k+1} \to \cdots \to \alpha_m \to \mathsf{o}$ or $\mathrm{type}(M) = \alpha_{k+1} \to \cdots \to \alpha_m \to \tau$ is replaced by $\lambda x_1{:}\,\alpha_1 \ldots \lambda x_m{:}\,\alpha_m.\,(M\, x_{k+1} \ldots x_m)$, until no further expansions are possible. Since η-expansions do not change the semantics of a query (cf. Section 5.3), P and Q are equivalent.

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

Lemma 7.1.3 *Let Q be a $TLI_i^=$ or $MLI_i^=$ term in canonical form mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then the following are true:*

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

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

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

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

5. *For $i \geq 1$, the only variables in Q of order i or more are $R_1, \ldots, R_l$ and accumulator variables (and c, if $i = 1$).*

Proof: This follows immediately from the type of Q and the fact that Q is canonical, i. e., fully η-expanded. □

7.2 Evaluating $TLI_0^=$ and $MLI_0^=$ Terms

$TLI_0^=/MLI_0^=$ queries can only iterate over order 0 objects. This means that the processing at each stage cannot depend on the value computed so far—the incoming value is a "black box" that has no externally visible features[1]. Intuitively, it should therefore be possible to perform the stages of an iteration *in parallel*, thus computing the output of a query in constant parallel time. This leads to the conjecture that $TLI_0^=/MLI_0^=$ queries are in AC_0, i. e., computable by constant-depth circuits with unbounded fan-in. We prove this conjecture using Immerman's characterization of uniform AC_0 as the first-order expressible properties, by showing that the output of a $TLI_0^=/MLI_0^=$ query can be described by a first-order formula.

First we sharpen Lemma 7.1.3 for $TLI_0^=$ and $MLI_0^=$ terms:

[1] This is not entirely true, since the incoming value could be a $TLC^=$ constant, which is meaningful to the Eq predicate. However, since Eq cannot be typed as $\mathsf{o} \to \mathsf{o} \to \mathsf{o} \to \mathsf{o} \to \mathsf{o}$, it cannot be used in selecting the value to pass on to the next stage.

Lemma 7.2.1 *Let Q be a $TLI_0^=$ or $MLI_0^=$ term in canonical form mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then every subterm of Q of type τ has one of the following forms:*

1. $R_i\,(\lambda x_1{:}\mathsf{o} \ldots \lambda x_{k_i}{:}\mathsf{o}.\, \lambda y{:}\tau.\, M)\, N$, *where M and N are terms of type τ,*
2. $Eq\, S\, T\, U\, V$, *where S and T are terms of type o and U and V are terms of type τ,*
3. $c\, T_1 \ldots T_k\, T_{k+1}$, *where $T_1, \ldots, T_k$ are terms of type o and T_{k+1} is a term of type τ,*
4. y, *where y is some variable of type τ.*

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

5. $R_i\,(\lambda x_1{:}\mathsf{o} \ldots \lambda x_{k_i}{:}\mathsf{o}.\, \lambda y{:}\mathsf{o}.\, M)\, N$, *where M and N are terms of type o,*
6. y, *where y is some variable of type o,*
7. o_j, *where o_j is a $TLC^=$ constant.*

Proof: Let M be a subterm of Q of type τ and let s be its top-level symbol, i. e., $M = s\, M_1 \ldots M_n$. s must be one of R_i, Eq, c, n, or an accumulator variable of type τ, since no other variables can occur in Q. For each of these cases, it follows from Lemma 7.1.3 and the type of Q that one of (1)–(4) must apply.

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

For a term Q as described in the above lemma, we will now construct a first-order formula $\psi_Q\,(\xi_1, \ldots, \xi_k)$ describing its behavior. More precisely, $\psi_Q\,(\xi_1, \ldots, \xi_k)$ is a formula with free variables $\xi_1, \ldots, \xi_k$ built from predicate symbols $R_1, \ldots, R_l$ of arities $k_1, \ldots, k_l$ and an interpreted predicate "$<$" specifying a total order on the domain, such that for any structure $\mathcal{S} = (D, r_1, \ldots, r_l)$ over $R_1, \ldots, R_l$ and any tuple $(x_1, \ldots, x_k) \in D^k$, we have $\mathcal{S} \vdash \psi_Q\,(x_1, \ldots, x_k)$ iff $(x_1, \ldots, x_k)$ is in the output of $(Q\, \overline{r_1} \ldots \overline{r_l})$, where the encodings $\overline{r_1}, \ldots, \overline{r_l}$ are chosen so that the tuples appear in lexicographical order as determined by "$<$".

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

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

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

To capture the output of an iteration in a first-order formula, we therefore need to express two things: (a) a stage of the iteration prepends tuples to the incoming value, i. e., it "passes through" all incoming tuples to the next stage, and (b) a stage of the iteration "produces" some tuple $(\xi_1, \ldots, \xi_k)$. This leads to the definition of two sets of formulas: a formula $\mathit{PassThrough}_{z,t}$ for every subterm $t{:}\,\tau$ of Q and variable $z{:}\,\tau$ free in t, saying that term t will "pass through" whatever tuples are in z to its output, and a formula $\mathit{Produces}_t\,(\xi_1, \ldots, \xi_k)$ for every subterm $t{:}\,\tau$ of Q saying that tuple $(\xi_1, \ldots, \xi_k)$ will be in the output of t. These formulas are defined below by structural induction over t. The formula $\mathit{Produces}_{Q'}\,(\xi_1, \ldots, \xi_k)$, where Q' is the "body" of $Q \equiv \lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$, is then the desired first-order equivalent of Q.

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

Here are the actual definitions of the *PassThrough* and *Produces* predicates. We use the notation $\vec{x} < \vec{y}$ to express that $\vec{x}$ lexicographically precedes $\vec{y}$.

$$
\begin{aligned}
\mathit{PassThrough}_{z,x} &\equiv \begin{cases} \mathit{False} & \text{if } z \not\equiv x \\ \mathit{True} & \text{if } z \equiv x \end{cases} \quad \text{(where } x{:}\,\tau \text{ is a variable)} \\
\mathit{PassThrough}_{z,cT_1\ldots T_k T_{k+1}} &\equiv \mathit{PassThrough}_{z,T_{k+1}} \\
\mathit{PassThrough}_{z,EqSTUV} &\equiv \exists\, x\, y{:}\, \mathit{Produces}'_S(x) \wedge \mathit{Produces}'_T(y) \\
&\quad \wedge \Big((x = y \wedge \mathit{PassThrough}_{z,U}) \\
&\qquad \vee (x \neq y \wedge \mathit{PassThrough}_{z,V})\Big) \\
\mathit{PassThrough}_{z,R_i(\lambda\vec{x}\lambda y.M)N} &\equiv \Big(\mathit{PassThrough}_{z,N} \wedge \forall \vec{x} \in R_i{:}\, \mathit{PassThrough}_{y,M}\Big) \\
&\quad \vee \Big(\exists \vec{x}_0 \in R_i{:}\, \mathit{PassThrough}_{z,M[\vec{x}:=\vec{x}_0]} \\
&\qquad \wedge \forall \vec{x} \in R_i, \vec{x} < \vec{x}_0{:}\, \mathit{PassThrough}_{y,M}\Big)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Produces}_x(\xi_1,\ldots,\xi_k) &\equiv \mathit{False} \quad \text{(where } x{:}\,\tau \text{ is a variable)} \\
\mathit{Produces}_{cT_1\ldots T_k T_{k+1}}(\xi_1,\ldots,\xi_k) &\equiv \left(\bigwedge_{i=1}^{k} \mathit{Produces}'_{T_i}(\xi_i)\right) \vee \mathit{Produces}_{T_{k+1}}(\xi_1,\ldots,\xi_k) \\
\mathit{Produces}_{EqSTUV}(\xi_1,\ldots,\xi_k) &\equiv \exists x\, y{:}\, \mathit{Produces}'_S(x) \wedge \mathit{Produces}'_T(y) \\
&\quad \wedge ((x = y \wedge \mathit{Produces}_U(\xi_1,\ldots,\xi_k)) \\
&\qquad \vee (x \neq y \wedge \mathit{Produces}_V(\xi_1,\ldots,\xi_k))) \\
\mathit{Produces}_{R_i(\lambda\vec{x}\lambda y.M)N}(\xi_1,\ldots,\xi_k) &\equiv \Big(\mathit{Produces}_N(\xi_1,\ldots,\xi_k) \wedge \forall \vec{x} \in R_i{:}\, \mathit{PassThrough}_{y,M}\Big) \\
&\quad \vee \Big(\exists \vec{x}_0 \in R_i{:}\, \mathit{Produces}_{M[\vec{x}:=\vec{x}_0]}(\xi_1,\ldots,\xi_k) \\
&\qquad \wedge \forall \vec{x} \in R_i, \vec{x} < \vec{x}_0{:}\, \mathit{PassThrough}_{y,M}\Big)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{PassThrough}'_{z,o_j} &\equiv \mathit{False} \quad \text{(where } o_j \text{ a constant)} \\
\mathit{PassThrough}'_{z,x} &\equiv \begin{cases} \mathit{False} & \text{if } z \not\equiv x \\ \mathit{True} & \text{if } z \equiv x \end{cases} \quad \text{(where } x{:}\,\mathsf{o} \text{ is a variable)} \\
\mathit{PassThrough}'_{z,R_i(\lambda\vec{x}\lambda y.M)N} &\equiv \Big(\mathit{PassThrough}'_{z,N} \wedge \neg\exists \vec{x} \in R_i\Big) \\
&\quad \vee \Big(\mathit{PassThrough}'_{z,N} \wedge \mathit{PassThrough}'_{y,M} \wedge \exists \vec{x} \in R_i\Big) \\
&\quad \vee \Big(\mathit{PassThrough}'_{z,M} \wedge \exists \vec{x} \in R_i\Big)
\end{aligned}
$$

$$
\mathit{Produces}'_{o_j}(\xi) \equiv (\xi = o_j) \quad \text{(where } o_j \text{ is a constant)}
$$

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

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

Lemma 7.2.2 *Let $Q \equiv \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ be a $TLI_0^=$ or $MLI_0^=$ term in canonical form, t be a subterm of Q of type τ, t' be a subterm of Q of type o, $\overline{r_1}, \ldots, \overline{r_l}$ be a legal input for Q with tuples appearing in lexicographical order, $FV_{\mathsf{o}}(t)$ be the set of free variables of t of type o, and ρ be a substitution that assigns constants to all variables in $FV_{\mathsf{o}}(t)$. Then the following are true:*

1. *For each free variable z of t of type τ, $PassThrough_{z,t}$ defines a first-order formula with free variables in $FV_{\mathsf{o}}(t)$.*

2. *$Produces_t(\xi_1, \ldots, \xi_k)$ defines a first-order formula with free variables in $FV_{\mathsf{o}}(t) \cup \{\xi_1, \ldots, \xi_k\}$.*

3. *For each free variable z of t' of type o, $PassThrough'_{z,t'}$ defines a first-order formula with no free variables.*

4. *$Produces_{t'}(\xi)$ defines a first-order formula with free variables in $FV_{\mathsf{o}}(t') \cup \{\xi\}$.*

5. *The expression $t\,[R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]\,\rho$ reduces to a list-like structure*

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

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

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

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

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

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

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

   In the second case, we have

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

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

Proof: By induction over the structure of subterms of Q and, in the case of a subterm $R_i\,(\lambda\vec{x}.\,\lambda y.\,M)\,N$, over the length of R_i. $\Box$

Theorem 7.2.3 *Under the input/output conventions of Section 6.1, $TLI_0^=$ and $MLI_0^=$ express exactly the first-order queries over ordered structures, provided that the input is presented in lexicographically sorted order.*

Proof: That $\mathrm{TLI}_0^=$ and $\mathrm{MLI}_0^=$ can express first-order queries and the ordering of their input was shown in Theorem 6.3.1. The upper bound follows from Lemma 7.2.2, because the output of a query term $Q \equiv \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ is described by the formula $Produces_{Q'}$.$\Box$

7.3 Evaluating $\mathbf{TLI_1^=}$ and $\mathbf{MLI_1^=}$ Terms

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

Lemma 7.3.1 *Let $Q = \lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ be a $TLI_1^=$ or $MLI_1^=$ term in canonical form mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then every complete subterm of Q' which is not the first argument of an occurrence of some R_i has the form $\lambda\vec{x}.\,M$, where $\vec{x}$ is a (possibly empty) vector of order-zero variables and M is a term of order 0 having one of the following forms:*

1. *$R_i\,(\lambda\vec{x}.\,\lambda f.\,M)\,N\,T_1\ldots T_m$, where $m \geq 0$, $\vec{x}$ is a vector of k_i variables of type* o*, the order of f, M, and N is at most 1, and the order of $T_1,\ldots,T_m$ is 0.*

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

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

4. *$f\,T_1\ldots T_m$, where $m \geq 0$, f is an accumulator variable and $T_1,\ldots,T_m$ are terms of order 0*

5. *x, where x is a variable of order 0 or a constant.*

Proof: According to Lemma 7.1.3, the only bound variables in Q' of non-zero order are accumulator variables, and these are bound in the first argument of occurrences of the R_i's. Any complete subterm $\lambda\vec{x}.\,M$ of Q' which is not the first argument of some R_i can therefore only abstract on order 0 variables.

Suppose now that s is the top-level symbol of M, i. e., $M = s\,M_1\ldots M_n$. There are five possibilities for s: it can be one of $R_1,\ldots,R_l$, in which case Lemma 7.1.3 and the type of Q imply that M is of form (1); it can be Eq, in which case form (2) applies; it can be c, in which case form (3) applies; it can be an accumulator variable, in which case form (4) applies; or it can be a variable of order 0 or a constant, in which case form (5) applies. □

7.3.1 The Evaluation Algorithm—An Overview

Before we plunge into details, let us first describe informally how a PTIME evaluation algorithm for TLI$_1^=$/MLI$_1^=$ terms might work.

The evaluation algorithm essentially has to deal with the five kinds of expressions listed in Lemma 7.3.1. Once $R_1,\ldots,R_l$ are instantiated, these expressions normalize to terms of the form $\lambda\vec{x}.\,M$, where $\vec{x}$ is a (possibly empty) vector of order 0 variables and M is a λ-free term of order 0 built from Eq, c, variables of order 0, and constants. Unfortunately, these normal forms can be of exponential size for two reasons: (1) an exponential number of occurrences of Eq or (2) an exponential number of occurrences of c. A PTIME evaluation algorithm must deal with these two situations.

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

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

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

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

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

where x is some variable of type τ. If such a term is of exponential size, then only because the list contains many duplicates. It is easy to see that elimination of these duplicates does not affect the output relation produced by a query, even though it may cause the computed representation of the output to be different (it will be the duplicate-free version of the original representation). Thus, the evaluation algorithm is free to remove duplicates from every term $M_{i_1,\ldots,i_m}$ it constructs, thereby always returning terms of polynomial size.

Using the above polynomial-size representation of order 1 terms, the evaluation of a canonical form query $\lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$ on input $\overline{r_1}, \ldots, \overline{r_l}$ now proceeds as a recursive descent into Q'. Subterms of the form $R_i\,(\lambda\vec{x}.\, \lambda f.\, M)\, N\, T_1 \ldots T_m$ are evaluated by evaluating N first and then evaluating the "loop body" M once for each tuple in r_i, from last to first, with $\vec{x}$ bound to the current tuple and f bound to the result of the previous iteration (in decision tree format). The final result of the loop is then applied to the evaluated values of $T_1 \ldots T_m$.

Subterms of the form $(Eq\, S\, T\, U\, V)$ or $(c\, T_1 \ldots T_{k+1})$ are evaluated by evaluating the arguments first and then constructing a decision tree for the result. Finally, subterms of the form $f\, T_1 \ldots T_m$ are evaluated by substituting the evaluated arguments (which must be order 0 terms) into the decision tree for f. We give the full details in the next section. It is easy to see that this procedure terminates after a number of steps polynomial in the size of $\overline{r_1}, \ldots, \overline{r_l}$ and that the work performed at each steps is polynomial as well.

7.3.2 The Evaluation Algorithm in Detail

In the following, let $Q = \lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$ be a fixed TLI$_1^=$ or MLI$_1^=$ query term in γ-canonical form, for some fixed canonical typing γ of Q. Whenever we refer to the type of a subterm t of Q in the following, we mean the canonical type of t under γ. In general, we adopt the "Church viewpoint" in the discussion below, where each term comes with a fixed type specified by type annotations on its free and bound variables.

We also fix a particular input $\overline{r_1}, \ldots, \overline{r_l}$ to Q and denote by D the set of TLC$^=$ constants appearing in $\overline{r_1}, \ldots, \overline{r_l}$, say $\{o_1, \ldots, o_N\}$. The task of the evaluation algorithm is then to produce the relation represented by the normal form of $\lambda c.\, \lambda n.\, Q'\,[R_1 {:=}\, \overline{r_1}, \ldots, R_l {:=}\, \overline{r_l}]$.

The free variables of $Q'\,[R_1 {:=}\, \overline{r_1}, \ldots, R_l {:=}\, \overline{r_l}]$ are c and n. In the subsequent discussion, it will be convenient to treat these variables, along with Eq and $o_1, \ldots, o_N$, as *constants* that are handled specially by the evaluation algorithm. Thus, if we speak of the set of free variables of some term t below, we mean the free variables of t *except* c and n.

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

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

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

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

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

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

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

Lemma 7.3.2 *Let* $t \sim_{\alpha\to\beta} t'$ *and* $s \sim_\alpha s'$. *Then* $\mathrm{nf}\,(t\,s) \sim_\beta \mathrm{nf}\,(t'\,s')$.

Proof: It suffices to show that $\mathrm{nf}\,(t\,s) \sim_\beta \mathrm{nf}\,(t\,s')$, because we have $\mathrm{nf}\,(t\,s') \sim_\beta \mathrm{nf}\,(t'\,s')$ by virtue of the hypothesis $t \sim_{\alpha\to\beta} t'$.

After performing an η-expansion, if necessary, we can assume that t is of the form $\lambda x{:}\,\alpha.\, f\, t_1 \ldots t_m$, where $f \in \{x, c, n, Eq, o_1, \ldots, o_N\}$ and $t_1, \ldots, t_m$ are terms. If $f \equiv n$ or

$f \in \{o_1, \ldots, o_N\}$, then clearly $t\,s \sim t\,s'$. Otherwise, we proceed by induction on the order of $\alpha \to \beta$ and, for fixed order, the size of $f\,t_1 \ldots t_m$.

Suppose that $f \equiv c$. Then $\mathrm{type}(t_1) = \cdots = \mathrm{type}(t_{m-1}) = \mathsf{o}$ and $\mathrm{type}(t_m) = \tau$. By the induction hypothesis, we have $\mathrm{nf}(t_i\,[x{:=}\,s]) \equiv \mathrm{nf}((\lambda x.\,t_i)\,s) \sim \mathrm{nf}((\lambda x.\,t_i)\,s') \equiv \mathrm{nf}(t_i\,[x{:=}\,s'])$ for $1 \le i \le m$. Thus, $t_i\,[x{:=}\,s]$ and $t_i\,[x{:=}\,s']$ reduce to the same constant for $1 \le i \le m-1$ and to representations of the same relation for $i = m$. It follows that $\mathrm{nf}(t\,s) \equiv c\,\mathrm{nf}(t_1\,[x{:=}\,s]) \ldots \mathrm{nf}(t_m\,[x{:=}\,s])$ and $\mathrm{nf}(t\,s') \equiv c\,\mathrm{nf}(t_1\,[x{:=}\,s']) \ldots \mathrm{nf}(t_m\,[x{:=}\,s'])$ represent the same relation as well, and hence $\mathrm{nf}(t\,s) \sim \mathrm{nf}(t\,s')$.

If $f \equiv Eq$, then $m = 4$ and $\mathrm{type}(t_1) = \mathrm{type}(t_2) = \mathsf{o}$, $\mathrm{type}(t_3) = \mathrm{type}(t_4) = \tau$. Using the same argument as above, it follows that for $1 \le i \le 4$, $t_i\,[x{:=}\,s]$ and $t_i\,[x{:=}\,s']$ reduce to the same constant or representations of the same relation, and hence the normal forms of $t\,s$ and $t\,s'$ represent the same relation, too.

If $f \equiv x$, then again the normal forms of $t_i\,[x{:=}\,s]$ and $t_i\,[x{:=}\,s']$ are equivalent, and by applying the inductive hypothesis iteratively to s and s', $\mathrm{nf}(s\,t_1\,[x{:=}\,s])$ and $(s'\,t_1\,[x{:=}\,s'])$, etc., we can deduce that $\mathrm{nf}(s\,t_1\,[x{:=}\,s] \ldots t_m\,[x{:=}\,s]) \sim \mathrm{nf}(s'\,t_1\,[x{:=}\,s'] \ldots t_m\,[x{:=}\,s'])$ and hence $\mathrm{nf}(t\,s) \sim \mathrm{nf}(t\,s')$. □

Since we are only interested in the equivalence class of the normal form of $Q'\,[R_1{:=}\,\overline{r_1}, \ldots, R_l{:=}\,\overline{r_l}]$, it suffices to manipulate equivalence classes of terms during the evaluation of the query, rather than actual λ-terms. Thus, an intermediate result t of type $\alpha_1 \to \cdots \to \alpha_m \to \beta$, where β is o or τ, can be stored as a table or decision tree describing the action of t on each vector of equivalence classes from $\overline{\mathcal{E}}_{\alpha_1} \times \cdots \times \overline{\mathcal{E}}_{\alpha_m}$, rather than the term t itself. If t contains free variables, say $y_1, \ldots, y_n$, these can be accommodated by storing the equivalence class of $\lambda y_1 \ldots \lambda y_n.\,t$ instead and noting the fact that $y_1, \ldots, y_n$ were free in the original term. (More formally, we extend the definition of $\sim_\gamma$ to terms containing free variables by saying that $t \sim_\gamma t'$ iff t and t' have the same set of free variables $\{y_1, \ldots, y_n\}$ and $\lambda y_1 \ldots \lambda y_n.\,t \sim_\gamma \lambda y_1 \ldots \lambda y_n.\,t'$.)

The trick of representing intermediate results as graphs rather than λ-terms imposes a fixed bound on the size of these representations, but unfortunately the bound is quite large. For example, there are exponentially many equivalence classes in $\overline{\mathcal{E}}_\tau$ (one for each k-ary relation over D), and thus a table describing an order 1 term, such as the "accumulator value" passed between stages of an iteration, would have to be of exponential size. Since we are shooting for a PTIME evaluation algorithm for TLI$_1^=$ terms, this is not good enough.

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

Lemma 7.3.3 *Let t and t' be terms in $\overline{\mathcal{E}}_{\tau\to\alpha}$ and e and f be terms in $\overline{\mathcal{E}}_\tau$ corresponding to the relations $\emptyset$ and D^k, respectively. If* $\mathrm{nf}\,(t\,e) \sim_\alpha \mathrm{nf}\,(t'\,e)$ *and* $\mathrm{nf}\,(t\,f) \sim_\alpha \mathrm{nf}\,(t'\,f)$*, then* $\mathrm{nf}\,(t\,r) \sim_\alpha \mathrm{nf}\,(t'\,r)$ *for every $r \in \overline{\mathcal{E}}_\tau$, i. e., $t \sim_{\tau\to\alpha} t'$.*

Proof: Let $\alpha = \alpha_1 \to \cdots \to \alpha_m \to \beta$, where $\beta = \mathsf{o}$ or $\beta = \tau$, and let $s_i \in \overline{\mathcal{E}}_{\alpha_i}$ for $1 \le i \le m$. Let T and T' be the normal forms of $\lambda x\!:\!\tau.\, t\,x\,s_1 \ldots s_m$ and $\lambda x\!:\!\tau.\, t'\,x\,s_1 \ldots s_m$, respectively. Assume that $\mathrm{nf}\,(T\,e) \sim_\beta \mathrm{nf}\,(T'\,e)$ and $\mathrm{nf}\,(T\,f) \sim_\beta \mathrm{nf}\,(T'\,f)$; we have to show that $T \sim_{\tau\to\beta} T'$.

The terms $t\,x\,s_1 \ldots s_m$ and $t'\,x\,s_1 \ldots s_m$ reduce to normal forms containing only the symbols $x, c, n, o_1, \ldots, o_N$ and having type either o or τ. It is easy to see that in the former case, the normal form must be a constant from $\{o_1, \ldots, o_N\}$ and in the latter case, the normal form must be a term

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

where $m \ge 0$ and y is either n or x. In the former case, the mappings defined by T and T' are constant, and since they agree on e and f, they agree everywhere. In the latter case, the mappings are either constant (if $y \equiv n$) or (if $y \equiv x$) "linear" in x, in the sense that the output consists of a fixed set of tuples S plus whatever tuples are in the value supplied for x. In both cases, the fact that they agree on two inputs implies that they agree everywhere.□

By virtue of Lemma 7.3.3, the equivalence class of an order 1 term t having p inputs of type o and q inputs of type τ can now indeed be stored in polynomial space. It suffices to record the output of t for each of the $N^p\, 2^q$ possible assignments of constants $o_1, \ldots, o_N$ to inputs of type o and relations $\emptyset, D^k$ to inputs of type τ. To recover the output of t for some input $(o_{i_1}, \ldots, o_{i_p}, r_1, \ldots, r_q)$ containing relations $r_1, \ldots, r_q$ other than $\emptyset$ or D^k, one proceeds as follows: First, the output of t on input $(o_{i_1}, \ldots, o_{i_p}, r_1, \emptyset, \ldots, \emptyset)$ is determined by looking up the output of t on input $(o_{i_1},\ldots,o_{i_p},\emptyset,\emptyset,\ldots,\emptyset)$ and, if the latter is different from the output of t on input $(o_{i_1},\ldots,o_{i_p},D^k,\emptyset,\ldots,\emptyset)$, adding the tuples in r_1 to it; then the same technique is used to determine the output of t for input $(o_{i_1},\ldots,o_{i_p},r_1,r_2,\emptyset,\ldots,\emptyset)$, and so forth, until finally the output of t for input $(o_{i_1},\ldots,o_{i_p},r_1,\ldots,r_q)$ is found.

We record for posteriority:

Lemma 7.3.4 *Let $\gamma = \alpha_1 \to \cdots \to \alpha_m \to \beta$ be a type of order $k+1$ in $\mathcal{T}$ and t be a term in $\overline{\mathcal{E}}_\gamma$. Let $[\![t]\!]$ denote the equivalence class modulo $\sim_\gamma$ of t. Then a description of $[\![t]\!]$ can be stored in space $e\,(k,N)$ such that for all inputs $[\![s_1]\!],\ldots,[\![s_m]\!]$ of the appropriate type, the equivalence class of $t\,s_1\ldots s_m$ can be determined in time $e\,(k,N)$. Here, N is the size of the input universe $\{o_1,\ldots,o_N\}$, $e\,(0,N)$ stands for $N^{O(1)}$, and $e\,(k+1,N)$ stands for $2^{e(k,N)}$.*

Proof: According to the discussion above, the equivalence classes of order 1 terms can be stored as a tables or decision trees in polynomial space. Since the graphs of equivalence classes of order $k+1$ are exponentially larger than the graphs of equivalence classes of order k, it follows that they can be stored in space $e\,(k,N)$. To determine the output of $[\![t]\!]$ for a particular set of inputs $[\![s_1]\!],\ldots,[\![s_m]\!]$, a table lookup or decision tree traversal suffices (plus the special processing for inputs of type τ described above), hence the time required is $e\,(k,N)$. □

We will now give a pseudo-code definition of the evaluation algorithm for $\mathrm{TLI}_1^=$ terms. The algorithm is just an ordinary λ-reduction engine, except that instead of returning the normal form of a term directly, it returns its equivalence class in decision tree format. For the purpose of specifying such decision trees, we adopt the following notation: If $[\![t]\!]$ is an equivalence class mapping inputs $x_1{:}\,\alpha_1,\ldots,x_m{:}\,\alpha_m$ to an output of type o or τ, we write $[\![t]\!]$ as the pseudo-$\mathrm{TLC}^=$ term

$$
\begin{array}{l}
[\![t]\!] \equiv \lambda x_1 \ldots \lambda x_m. \\
\qquad Eq\, x_1\, [\![a_{1,1}]\!]\, (Eq\, x_2\, [\![a_{2,1}]\!] \ldots (Eq\, x_m\, [\![a_{m,1}]\!]\, s_{1,1,\ldots,1,1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (Eq\, x_m\, [\![a_{m,2}]\!]\, s_{1,1,\ldots,1,2} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (Eq\, x_m\, [\![a_{m,N_m-1}]\!]\, s_{1,1,\ldots,1,N_m-1}\, s_{1,1,\ldots,1,N_m}))\ldots) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad (Eq\, x_2\, [\![a_{2,2}]\!] \ldots (Eq\, x_m\, [\![a_{m,1}]\!]\, s_{1,2,\ldots,1,1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (Eq\, x_m\, [\![a_{m,2}]\!]\, s_{1,2,\ldots,1,2}
\end{array}
$$

$$\begin{array}{lll} & & \vdots \\ & & (Eq\, x_m\, [\![a_{m,N_{m-1}}]\!]\, s_{1,2,\ldots,1,N_{m-1}}\, s_{1,2,\ldots,1,N_m}))\ldots) \\ & \vdots & \vdots \\ Eq\, x_1\, [\![a_{1,2}]\!] & (Eq\, x_2\, [\![a_{2,1}]\!] \ldots & (Eq\, x_m\, [\![a_{m,1}]\!]\, s_{2,1,\ldots,1,1} \\ \vdots & \vdots & \vdots \end{array}$$

Here, $\{[\![a_{i,j}]\!] \mid 1 \le j \le N_i\}$ is an enumeration of the equivalence classes in $\overline{\mathcal{E}}_{\alpha_i}$ in some standard order (if $\alpha_i = \tau$, only $\emptyset$ and D^k are used), and the terms $s_{i_1,\ldots,i_m}$ are either constants or relations depending on the output type of t. Note that this tree is *not* a TLC$^=$ term—it is just a human-readable representation of some internal data structure used by the interpreter. In particular, the symbol *Eq* above is not the TLC$^=$ symbol $Eq{:}\, \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$; rather, it is an instruction to the interpreter to compare two equivalence classes and choose the appropriate branch in the tree. With these precautions in mind, however, it is convenient to think of $[\![t]\!]$ as if it were a TLC$^=$ term, and we will occasionally blur the distinction in the discussion below.

If $[\![s_1]\!], \ldots, [\![s_m]\!]$ are equivalence classes in $\overline{\mathcal{E}}_{\alpha_1}, \ldots, \overline{\mathcal{E}}_{\alpha_m}$, we write $[\![t]\!]\, [\![s_1]\!] \ldots [\![s_m]\!]$ to denote the equivalence class of $t\, s_1 \ldots s_m$; the interpreter finds the latter by walking down the decision tree for $[\![t]\!]$ and picking the leaf determined by $[\![s_1]\!], \ldots, [\![s_m]\!]$ (for inputs of type τ, the linearity processing described before Lemma 7.3.4 is performed).

The evaluation algorithm consists of a recursive procedure $\mathit{Eval}(t)$ that takes a term $t \in \overline{\mathcal{E}}_\gamma$ containing no free variables other than $R_1, \ldots, R_l$ and produces the equivalence class of $\mathrm{nf}\,(t\,[R_1{:=}\, \overline{r_1}, \ldots, R_l{:=}\, \overline{r_l}])$. To simplify the presentation, we also allow occurrences of pseudo-TLC$^=$ terms $[\![s]\!]$ in t; such an occurrence is treated by the interpreter as if it were the TLC$^=$ term s (except that evaluation is much faster because the interpreter knows the behavior of s). In fact, we assume that occurrences of the TLC$^=$ symbols Eq, c, and n in t have been replaced by pseudo-TLC$^=$ terms describing their equivalence classes, e. g., t contains $\emptyset$ instead of n and $[\![\lambda x{:}\mathsf{o}.\, \lambda y{:}\mathsf{o}.\, \lambda u{:}\, \tau.\, \lambda v{:}\, \tau.\, Eq\, x\, y\, u\, v]\!]$ instead of Eq. Due to this modification, the five possible subterms listed in Lemma 7.3.1 combine to three possible forms of t that we need to consider: (a) $t \equiv [\![T]\!]\, S_1 \ldots S_m$, where $[\![T]\!]$ is a pseudo-TLC$^=$ term, (b) $t \equiv R_i\,(\lambda \vec{x}.\, \lambda f.\, M)\, N\, T_1 \ldots T_m$, and (c) $t \equiv \lambda x{:}\, \alpha.\, M$, where α is o or τ. For these cases, *Eval* is defined as follows.

$$\mathit{Eval}\,([\![t]\!]\, s_1 \ldots s_m) \equiv$$

$$[\![s_i]\!] = \mathit{Eval}\,(s_i), \quad \text{for } 1 \le i \le m$$

return $[\![t]\!]\,[\![s_1]\!]\ldots[\![s_m]\!]$

$Eval\,(R_i\,(\lambda\vec{x}.\,\lambda f.\,M)\,N\,T_1\ldots T_m) \equiv$

$A = Eval\,(N)$

for every tuple $\vec{o} \in r_i$ from last to first do

$A = Eval\,(M\,[\vec{x}{:=}\,\vec{o}, f{:=}\,A])$

od

$[\![T_i]\!] = Eval\,(T_i)$, for $1 \leq i \leq m$

return $A\,[\![T_1]\!]\ldots[\![T_m]\!]$

$Eval\,(\lambda x{:}\,\gamma.\,M) \equiv$

for every equivalence class $[\![s_i]\!]$ in $\overline{\mathcal{E}}_\gamma$ do (if $\gamma = \tau$, only $\emptyset$ and D^k are considered)

$[\![M_i]\!] = Eval\,(M\,[x{:=}\,[\![s_i]\!]])$

od

return $\lambda x.\,Eq\,x\,[\![s_1]\!]\,[\![M_1]\!]\,(Eq\,x\,[\![s_2]\!]\,[\![M_2]\!]\ldots(Eq\,x\,[\![s_{N_\gamma-1}]\!]\,[\![M_{N_\gamma-1}]\!]\,[\![M_{N_\gamma}]\!]))\ldots)$

This concludes the specification of the evaluation algorithm. We now have to show that *Eval* is well defined, that it produces the correct output, and that it runs in polynomial time. This will be done in the following lemmas. We consider a fixed computation originating from a call $Eval\,(Q')$, where Q' is the "body" of a query term $Q = \lambda R_1\ldots\lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ in canonical form, $R_1,\ldots,R_l$ are bound to a legal input $\overline{r_1},\ldots,\overline{r_l}$, and $D = \{o_1,\ldots,o_N\}$ is the set of constants appearing in $\overline{r_1},\ldots,\overline{r_l}$.

Lemma 7.3.5 *For every invocation $Eval\,(t)$, t is in a form for which Eval is defined.*

Proof: An inspection of the algorithm shows that *Eval* is called only on terms t that are derived from complete subterms of Q' by replacing constants and free variables by pseudo-TLC$^=$ terms $[\![s]\!]$ and possibly stripping off some initial λ-abstractions. Lemma 7.3.1 implies that such terms can only take the following three forms: (a) $t \equiv [\![T]\!]\,S_1\ldots S_m$, (b) $t \equiv R_i\,(\lambda\vec{x}.\,\lambda f.\,M)\,N\,T_1\ldots T_m$, and (c) $t \equiv \lambda x{:}\,\alpha.\,M$. For each of these cases, *Eval* is defined. □

Lemma 7.3.6 *The decision tree returned by an invocation $Eval\,(t)$ is a representation of the equivalence class of $t\,[R_1{:=}\,\overline{r_1},\ldots,R_l{:=}\,\overline{r_l}]$. In particular, the call $Eval\,(Q')$ produces the equivalence class of the output of Q on inputs $r_1,\ldots,r_l$.*

Proof: By induction along the recursion tree. If $t \equiv [\![T]\!]\, S_1 \ldots S_m$ or $t \equiv \lambda x{:}\, \alpha.\, M$, the claim follows immediately from the definition of $[\![t]\!]$. If $t \equiv R_i\,(\lambda \vec{x}.\, \lambda f.\, M)\, N\, T_1 \ldots T_m$, an induction on the number of loop iterations shows that after the k^{th} iteration, the variable A contains the equivalence class of $(R_i^k\,(\lambda \vec{x}.\, \lambda f.\, M)\, N)\,[R_1{:}= \overline{r_1}, \ldots, R_l{:}= \overline{r_l}]$, where R_i^k contains the last k tuples of R_i. Thus, once the loop terminates, A contains the equivalence class of $(R_i\,(\lambda \vec{x}.\, \lambda f.\, M)\, N)\,[R_1{:}= \overline{r_1}, \ldots, R_l{:}= \overline{r_l}]$ and the claim follows. □

Lemma 7.3.7 *The number of recursive calls generated by an invocation Eval(t) and the work performed at each call is polynomial in the size of $r_1, \ldots, r_l$.*

Proof: The depth of the recursion tree generated by an invocation *Eval*(t) is bounded by the number of symbols in t (where pseudo-TLC$^=$ terms are counted as one symbol), because *Eval* calls itself only on proper subterms of its argument. The fanout at each node is at most polynomial in the size of $r_1, \ldots, r_l$, thus the total number of calls to *Eval* is polynomial as well. The work performed at each call is essentially proportional to the size of the data structures maintained. These are decision trees representing equivalence classes of certain fixed types of order 1 and hence are of size polynomial in the size of the universe. □

Corollary 7.3.8 *Database queries defined by terms in $TLI_1^=$ and $MLI_1^=$ can be evaluated in PTIME.*

Note that the definition of *Eval* and the proofs of Lemmas 7.3.5 and 7.3.6 do not depend on the order of the query term under consideration. If *Eval* is invoked on higher-order query terms, the depth of the recursion tree is still fixed, but the fanout and work performed at each node is now proportional to the size of the largest data structure maintained. Since the decision trees representing equivalence classes of order $k+1$ fit into k-EXPSPACE, one obtains the following rough upper bound on the expressiveness of $\text{TLI}_k^=$ and $\text{MLI}_k^=$:

Corollary 7.3.9 *Database queries defined by terms in $TLI_k^=$ and $MLI_k^=$ can be evaluated in k-EXPTIME.*

However, we will see in the following sections that we can do better than that.

Combining Corollary 7.3.8 with the encoding of fixpoint queries in $\text{TLI}_1^=$ presented in Section 6.3.3, we obtain:

Theorem 7.3.10 *Under the input/output conventions of Section 6.1, $TLI_1^=$ and $MLI_1^=$ express exactly the PTIME queries.*

Note that because TLI and MLI queries can discern the ordering of the tuples in the input encoding (cf. Section 6.3.1), $\text{TLI}_1^=$ and $\text{MLI}_1^=$ express *all* PTIME queries, not just the generic ones.

7.4 Evaluating $\mathrm{TLI}_2^=$ and $\mathrm{MLI}_2^=$ Terms

$\mathrm{TLI}_2^=/\mathrm{MLI}_2^=$ queries differ from $\mathrm{TLI}_1^=/\mathrm{MLI}_1^=$ queries in that they can pass order 2 terms from one stage of an iteration to the next. Such terms are powerful enough to represent arbitrary mappings from relations to relations (using the characteristic function representation of a relation). Thus, there is no hope of storing them in polynomial space. If we want to reduce $\mathrm{TLI}_2^=/\mathrm{MLI}_2^=$ queries in polynomial space nevertheless, we have to be careful about how we evaluate iterations.

The key idea is to evaluate an order 2 expression t only when it is applied to sufficiently many arguments to bring the overall order down to 1. Since we can represent order 1 expressions in polynomial space, using the "decision-tree" format described in the last section, space usage is kept within polynomial bounds. The tradeoff is that t might have to be evaluated many times, but we will see that the stack space required to do the proper bookkeeping is nevertheless polynomial.

Lemma 7.4.1 *Let $Q = \lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$ be a $TLI_2^=$ or $MLI_2^=$ term in canonical form mapping l relations of arities $k_1, \ldots, k_l$ to a relation of arity k. Then every complete subterm of Q' which is not the first argument of an occurrence of some R_i has the form $\lambda \vec{x}.\, M$, where $\vec{x}$ is a (possibly empty) vector of order-one and order-zero variables and M is a term of order 0 having one of the following forms:*

1. *$R_i\,(\lambda \vec{x}.\, \lambda F.\, \lambda g_1 \ldots \lambda g_m.\, M)\,(\lambda g_1 \ldots \lambda g_m.\, N)\, T_1 \ldots T_m$, where $m \geq 0$, $\vec{x}$ is a vector of k_i variables of type o, F is a variable of order at most 2, $g_1, \ldots, g_m$ are variables of order at most 1, M and N are terms of order 0, and $T_1, \ldots, T_m$ are terms of the same type as $g_1, \ldots, g_m$*
2. *$\mathrm{Eq}\, S\, T\, U\, V$, where S and T are terms of type o and U and V are terms of type τ,*
3. *$c\, T_1 \ldots T_k\, T_{k+1}$, where $T_1, \ldots, T_k$ are terms of type o and T_{k+1} is a term of type τ,*
4. *$F\, T_1 \ldots T_m$, where $m \geq 0$, F is an accumulator variable and $T_1, \ldots, T_m$ are terms of order at most 1*
5. *$g\, T_1 \ldots T_m$, where $m \geq 0$, g is a variable of order 1 bound in the first or second argument of an occurrence of some R_i, and $T_1, \ldots, T_m$ are terms of order 0*
6. *x, where x is a variable of order 0 or a constant.*

Proof: Analogous to Lemma 7.3.1. □

Lemma 7.4.1 shows that an evaluation algorithm for TLI$_2^=$/MLI$_2^=$ terms must handle three new cases as compared with TLI$_1^=$/MLI$_1^=$ terms: (a) subterms $\lambda\vec{g}.\,M$ abstracting on order 1 variables, (b) iterations $R\,(\lambda\vec{x}.\,\lambda F.\,\lambda\vec{g}.\,M)\,(\lambda\vec{g}.\,N)\,\vec{T}$ where the order of F is 2, and (c) applications $F\,T_1\ldots T_m$ where F is an order 2 variable. The other cases involve only order 1 terms and can be handled by the PTIME evaluation algorithm of Section 7.3.

We will see that we will never attempt to evaluate a term $\lambda\vec{g}.\,M$ abstracting on order 1 variables directly, so it suffices to explain the handling of cases (b) and (c). They are dealt with as follows. Fix a particular iteration expression

$$\mathcal{I} \equiv R\,(\lambda\vec{x}.\,\lambda F.\,\lambda g_1\ldots\lambda g_m.\,M)\,(\lambda g_1\ldots\lambda g_m.\,N)\,T_1\ldots T_m,$$

where F is of order 2. By the time *Eval* is called on $\mathcal{I}$, all free variables of $\mathcal{I}$ of order 1 or less will have been replaced by pseudo-TLC$^=$ terms denoting their equivalence classes. Thus, the only variables which may occur free in $\mathcal{I}$ are the input variables $R_1,\ldots,R_l$ and accumulator variables of order 2. To handle the latter, we provide an additional argument to *Eval*: an environment E containing an integer value for each accumulator variable of order 2 in whose scope the current term appears. This environment is initially empty and it is not used in any way by the order 1 cases of *Eval*. While evaluating the iteration denoted by $\mathcal{I}$, the integer bound to the accumulator variable F counts the number of iterations "still to go"; initially it is set to the number of tuples in R less one and the "loop body" M is evaluated with $\vec{x}$ substituted by the first tuple in R and $\vec{g}$ substituted by the equivalence classes of $T_1,\ldots,T_m$. If during the evaluation of M, an expression $F\,T_1'\ldots T_m'$ is encountered, the value bound to F is decremented by 1 and M is evaluated recursively with $\vec{x}$ substituted by the next tuple in R and $\vec{g}$ substituted by the equivalence classes of $T_1',\ldots,T_m'$. However, if F was bound to 0, then the "loop initializer" N is evaluated instead, with $\vec{g}$ substituted by the equivalence classes of $T_1',\ldots,T_m'$. More precisely, we have the following pseudo-code:

$\mathit{Eval}\,(R_i\,(\lambda\vec{x}.\,\lambda F.\,\lambda g_1\ldots\lambda g_m.\,M)\,(\lambda g_1\ldots\lambda g_m.\,N)\,T_1\ldots T_m, E) \equiv$

  $[\![T_i]\!] = \mathit{Eval}\,(T_i, E)$, for $1 \le i \le m$

  if $r_i = \emptyset$ then

    return $\mathit{Eval}\,(N\,[\vec{g} := [\![\vec{T}]\!]], E)$

  else

    $\vec{o} =$ the 1^{st} tuple in r_i

    return $\mathit{Eval}\,(M\,[\vec{x} := \vec{o}, \vec{g} := [\![\vec{T}]\!]], E \cup \{F := |r_i| - 1\})$

  fi

$Eval\,(F\,T_1 \ldots T_m, E) \equiv$

$[\![T_i]\!] = Eval\,(T_i, E), \quad$ for $1 \leq i \leq m$

$k =$ the binding of F in E

if $k = 0$ then

return $Eval\,(N\,[\vec{g} := [\![\vec{T}]\!]], E)$

else

$k = k - 1$

$\vec{o} =$ the $|r_i| - k^{\text{th}}$ tuple in r_i

return $Eval\,(M\,[\vec{x} := \vec{o}, \vec{g} := [\![\vec{T}]\!]], E \cup \{F := k\})$

fi

We now have to show that with these two additional cases, *Eval* is still correct and that it runs in polynomial space. This will be done in the following lemmas. We consider again a fixed computation originating from a call $Eval\,(Q', \emptyset)$, where Q' is the "body" of a query term $Q = \lambda R_1 \ldots \lambda R_l.\, \lambda c.\, \lambda n.\, Q'$ in canonical form, $R_1, \ldots, R_l$ are bound to a legal input $\overline{r_1}, \ldots, \overline{r_l}$, and $D = \{o_1, \ldots, o_N\}$ is the set of constants appearing in $\overline{r_1}, \ldots, \overline{r_l}$.

Lemma 7.4.2 *For every invocation $Eval\,(t, E)$, t is in a form for which Eval is defined and E has bindings for all order 2 accumulator variables whose scope includes t.*

Proof: An inspection of the algorithm shows that *Eval* is called only on terms t that are derived from complete subterms of Q' by replacing constants and free variables of order ≤ 1 by pseudo-TLC$^=$ terms $[\![s]\!]$ and possibly stripping off some initial λ-abstractions. Lemma 7.4.1 implies that such terms can only take the following four forms: (a) $t \equiv [\![T]\!]\,S_1 \ldots S_m$, (b) $t \equiv R_i\,(\lambda\vec{x}.\,\lambda f.\,M)\,N\,T_1 \ldots T_m$, (c) $t \equiv F\,T_1 \ldots T_m$, where $\text{order}\,(F) = 2$, and (d) $t \equiv \lambda x{:}\,\alpha.\,M$, where $\text{order}\,(\alpha) = 0$. For each of these cases, *Eval* is defined. Also, an inspection of the algorithm shows that a binding for an order 2 accumulator variable is added to the environment whenever the scope of that variable is entered. □

Lemma 7.4.3 *The decision tree returned by an invocation $Eval\,(t, E)$ is a representation of the equivalence class of a closed term t' obtained from t as follows. Let $F_1, \ldots, F_k$ be the order 2 accumulator variables whose scope includes t, listed in increasing order of nesting, and let $n_1, \ldots, n_k$ be the integers assigned to these variables in E. For $1 \leq j \leq k$, let*

$\mathcal{I}_j \equiv R_{i_j}(\lambda\vec{x}.\lambda F_j.M_j)N_j$ be the iterator expression in which F_j is bound. Thus, $\mathcal{I}_1$ is a subterm of Q', $\mathcal{I}_2$ is a subterm of M_1, $\mathcal{I}_3$ is a subterm of M_2 etc., and t is a subterm of M_k. Let $\overline{r_{i_j}}'$ denote the list containing the last n_j tuples of the list $\overline{r_{i_j}}$, let $\mathcal{I}'_1$ denote the term $(\overline{r_{i_1}}'(\lambda\vec{x}.\lambda F_1.M_1)N_1)[R_1{:=}\,\overline{r_1},\ldots,R_l{:=}\,\overline{r_l}]$, and for $2 \leq j \leq k$, let $\mathcal{I}'_j$ denote the term $(\overline{r_{i_j}}'(\lambda\vec{x}.\lambda F_j.M_j)N_j)[R_1{:=}\,\overline{r_1},\ldots,R_l{:=}\,\overline{r_l},F_1{:=}\,\mathcal{I}'_1,\ldots,F_{j-1}{:=}\,\mathcal{I}'_{j-1}]$. Then the decision tree returned by Eval(t,E) is a representation of the equivalence class of $t' \equiv t[R_1{:=}\,\overline{r_1},\ldots,R_l{:=}\,\overline{r_l},F_1{:=}\,\mathcal{I}'_1,\ldots,F_k{:=}\,\mathcal{I}'_k]$. In particular, the call Eval$(Q',\emptyset)$ produces the equivalence class of the output of Q on inputs $r_1,\ldots,r_l$.

Proof: By induction along the recursion tree. □

Lemma 7.4.4 *The recursion tree generated by the initial invocation Eval$(Q',\emptyset)$ is of polynomial depth and polynomial fanout. Also, the data structures stored at each node are of polynomial size.*

Proof: Let $F_1,\ldots,F_k$ be the order 2 accumulator variables occurring in Q', listed in order of nondecreasing nesting (i. e., for $i < j$, the scopes of F_i and F_j are either disjoint or the scope of F_i contains the scope of F_j). Let R_{i_j} be the symbol governing the iteration expression in which F_j is bound, and let N_j be the number of tuples in the input relation $\overline{r_{i_j}}$. Then the values bound to F_j during the course of the computation range over the interval $I_j = \{0,1,\ldots,N_j\}$.

With each call *Eval*(t,E), we associate a tuple $(n_1,\ldots,n_k,s) \in I_1\times\cdots\times I_k\times\{1,\ldots,|Q'|\}$ of integers as follows: s is defined as the number of symbols in t, counting pseudo-TLC$^=$ terms as one symbol. n_j is defined to be N_j if t is not in the scope of F_j and the integer bound to F_j in E otherwise. The tuple associated with the initial call *Eval*$(Q',\emptyset)$ is $(N_1,\ldots,N_k,|Q'|)$, and it is easy to see that each recursive call to *Eval* decreases at least one component of this tuple, leaving the components to the left unchanged and possibly increasing the components to the right. Since this can be done at most a polynomial number of times before bringing the term size down to 1, it follows that the recursion nesting depth must be polynomial. The fanout factor is dominated by the order 1 cases of *Eval* and is polynomial as seen before. Finally, the size of the environment and the equivalence classes manipulated at each call is polynomial as well. □

Corollary 7.4.5 *Database queries defined by terms in $TLI_2^=$ and $MLI_2^=$ can be evaluated in PSPACE.*

Combining this result with the generic simulation of Section 6.3.4, we obtain:

Theorem 7.4.6 *Under the input/output conventions of Section 6.1, $TLI^{=}_{2}$ and $MLI^{=}_{2}$ express exactly the PSPACE queries.*

7.5 Evaluating $\mathbf{TLI}^{=}_{3}$ and $\mathbf{MLI}^{=}_{3}$ Terms

The generalization of the evaluation strategy to $\mathrm{TLI}^{=}_{3}$ and $\mathrm{MLI}^{=}_{3}$ is straightforward. Since we want only an EXPTIME bound, we can allocate exponential space for the storage of intermediate results. This allows us to store the equivalence classes of terms up to order 2, and to evaluate iterations over order 0, 1, and 2 objects in applicative order, i. e., using the *Eval* procedures listed in Section 7.3. Iterations over order 3 objects are handled using the strategy of the previous section. The proofs of Lemmas 7.4.2—7.4.4 carry over to this new setting unchanged, except that order 2 has to be replaced by order 3 and that the recursion fanout factor and the data structure size may now be exponential. Nevertheless, since the recursion depth is still polynomial, it follows that the total number of calls is at most exponential and that the total work performed is at most exponential as well. Thus, together with the lower bound of Theorem 6.3.4, it follows that:

Theorem 7.5.1 *Under the input/output conventions of Section 6.1, $TLI^{=}_{3}$ and $MLI^{=}_{3}$ express exactly the EXPTIME queries.*

In general, to evaluate queries in $\mathrm{TLI}^{=}_{2k+1}$, $\mathrm{MLI}^{=}_{2k+1}$, $\mathrm{TLI}^{=}_{2k+2}$, and $\mathrm{MLI}^{=}_{2k+2}$, we use applicative order evaluation for subterms of order up to $k+1$, representing their normal forms as tables of size up to k-EXPSPACE, and normal order evaluation for subterms of higher order. We conjecture that for $\mathrm{TLI}^{=}_{2k+1}$ and $\mathrm{MLI}^{=}_{2k+1}$ queries, this leads to a time bound of k-EXPTIME, and for $\mathrm{TLI}^{=}_{2k+2}$ and $\mathrm{MLI}^{=}_{2k+2}$ queries, to a space bound of k-EXPSPACE, but we have not worked out the details in full yet.

Chapter 8

Method Schemas (MS)

Method schemas [4] are a simple programming formalism for OODBs, based on applicative program schemas with additional features such as *classes*, *methods*, *inheritance*, *name overloading*, and *late binding*. They are intended to model the object/class/method paradigm of object-oriented programming in the same way that applicative program schemas [17, 21] model traditional programming languages. In the next two sections, we describe the syntax and semantics of method schemas and we show that over ordered databases, they express exactly the PTIME queries.

8.1 Basic Definitions

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

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

A *k-term* for some $k \geq 1$ is a finite rooted ordered directed acyclic graph (dag) such that: (1) each vertex is uniquely labeled either by a method or by an integer in $\{1, \ldots, k\}$; (2) each vertex labeled by a method of arity j has j children which are ordered from left

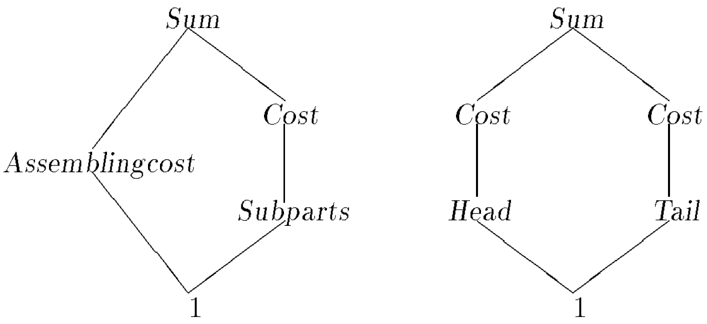

Figure 8.1: Term t and t'

to right; (3) all vertices of out-degree zero are called *inputs* and each vertex labeled with an integer is an input; (4) the root is called the *output*; (5) the dag comes with a uniquely defined depth first search order going from left to right because the children of each vertex are ordered.

For example, the terms t and t' in Figure 8.1 are 1-terms. They are, by definition, also 2-terms, 3-terms, etc. The idea is that k-terms represent functions of k arguments, where a copy of argument i is placed at each input labeled i. Vertices can also have outdegree zero and be labeled by zero-ary methods, i. e., they are constant inputs.

As an example, consider the following definitions:

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

Let s be a single arc with tail labeled *Price* and head labeled 1, and t be as in Figure 8.1. These definitions can be represented syntactically using an intuitive lambda notation—instead of the graphical k-term notation—as shown below. Note that we use the symbol colon (:) for representing base methods and the symbol equal (=) for coded methods.

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

Definition 8.1.1 *A* method schema *is a 4-tuple* $(C, \leq, \Sigma_0, \Sigma_1)$ *where:*

1. *C is a finite set of classes and $\leq$ is a forest on C. We say that $c \leq c'$ if c is a descendant of c' in this forest. The partial order defined by $\leq$ is called the* Isa *relation.*

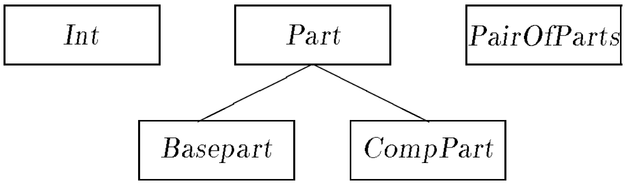

Figure 8.2: An example *Isa* hierarchy

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

Figure 8.3: Methods

2. *Σ_0 is a set of base definitions with classes from C.*

3. *Σ_1 is a set of coded definitions with classes from C. Let M_0 be the set of methods defined in Σ_0 and M_1 be the set of methods defined in Σ_1. We require that $M_0 \cap M_1 = \emptyset$ and that the methods that appear in k-terms of Σ_1 are from $M_0 \cup M_1$.*

4. *Each method of arity k has at most one definition at $c_1, \ldots, c_k$ for each $c_1, \ldots, c_k$.*

Example 8.1.2 *Consider the class hierarchy of Figure 8.2 and the terms t and t' in Figure 8.1. Method definitions are presented in Figure 8.3.*

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

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

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

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

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

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

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

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

Example 8.1.3 *Let c_1, c_2 be classes with $c_1 \leq c_2$ and consider the schema with explicit base definitions of m at c_1, c_2 and c_2, c_1. Then m is undefined at c_2, c_2 (no definition) and m is undefined at c_1, c_1 (ambiguity of definitions). For methods of arity 1, ambiguity (2) cannot arise because of the forest hierarchy, but we can have non-definition because of (1).*

8.2 Semantics

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

Definition 8.2.1 *Given a method schema* $(C, \leq, \Sigma_0, \Sigma_1)$*, a* disjoint object assignment ν *for* C *is a total function from* C *to finite sets of objects such that distinct classes are mapped to disjoint sets of objects (sets with pairwise empty intersections). For each* c*, we define* $\nu(c_*) = \bigcup_{c_1 \leq c} \nu(c_1)$*. We will refer to* $\bigcup_{c \in C} \nu(c)$ *as the set of objects in* ν*.*

Definition 8.2.2 *An* interpretation *of a schema* $S = (C, \leq, \Sigma_0, \Sigma_1)$ *is a pair* $I = (\nu, \mu)$ *where* ν *is a disjoint object assignment for* C *and* μ *a total function from the base methods* M_0 *in* S *to partial functions such that:*

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

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

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

Late binding: This is a result of the operational definition of rewriting. Let us first give some intuition for the rewriting of coded methods. Consider a k-term that is a single arc. Let its one internal node be labeled *Cost* and its input be replaced by an object o, as a "procedure call" to *Cost*. Based on the class of its argument, we can replace *Cost* either

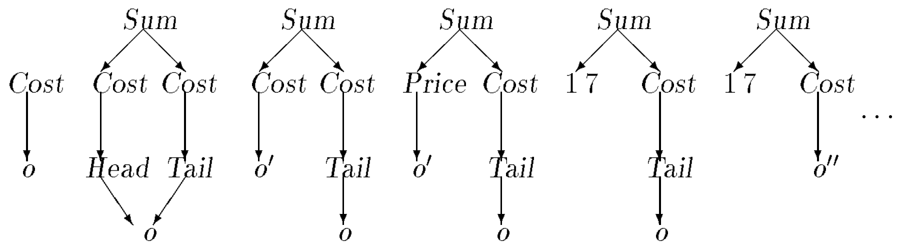

Figure 8.4: A reduction sequence

by some object/code if it is defined in a base/coded fashion, or by an error message if it is undefined. In general, we reduce the first method (first in the depth first ordering of the k-term we are processing) that has instantiated leaves as children. We continue this processing until we either obtain a result (i. e., an object), or reach an inconsistency. We might also not terminate. A partial sequence of rewritings for Example 8.1.2 is shown in Figure 8.4, where o is in class *PairOfParts* and o', o'' are in class *Basepart*.

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

Definition 8.2.3 *Let I be an interpretation of method schema S, t be an instantiated term in I, w be the first reducible vertex of t, and m be the label of w. Let $o_1, \ldots, o_k$ be the labels of the children of w belonging respectively to classes $c_1, \ldots, c_k$. The* reduction $I(t)$ *of t in I is obtained as follows:*

1. *If m is undefined at $c_1, \ldots, c_k$ then $I(t)$ is a special symbol $\perp$.*

2. *If m is a base method well defined at $c_1, \ldots, c_k$ with resolution $(m, (c'_1, \ldots, c'_k \to c))$ (this implies that $c_i \leq c'_i$ for $i = 1, \ldots, k$), then $I(t)$ is the instantiated term obtained by removing from w the outgoing edges and changing its label to $\mu(m)(o_1, \ldots, o_k)$. $\mu(m)$ is a total function from $\nu(c_1) \times \cdots \times \nu(c_k)$ into $\nu(c_*)$. Therefore, $\mu(m)(o_1, \ldots, o_k)$ yields an object o such that $o \in \nu(c_*)$.*

3. *If m is a coded method well defined at $c_1, \ldots, c_k$, then $I(t)$ is the instantiated term obtained by substituting, with the natural renaming of inputs, the vertex w by the dag s, where (m, σ, s) is the resolution of m at $c_1, \ldots, c_k$.*

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

8.3 The Expressive Power of Method Schemas

When considering method schemas in the context of finite databases, it is natural to ask what kind of queries can be expressed in the method schema framework. Of course, since method schemas are just *schemas*, one has to agree on the interpretation of the base methods in order to make this question precise. In this section, we describe a very simple way of encoding relations as interpreted base methods, and we show that under this interpretation, method schemas express exactly the PTIME queries. We assume some familiarity with Datalog terminology and notation from [30, 47], e. g., rules, EDBs, IDBs etc.

8.3.1 I/O Conventions

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

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

A k-ary relation R is represented by a k-ary base method

$$r \text{ @ } \mathit{Number}, \ldots, \mathit{Number}\colon \mathit{Number},$$

such that $r(x_1, \ldots, x_k)$ returns an object of class *Number* if the tuple represented by $(x_1, \ldots, x_k)$ belongs to R and an object of class *Zero* if it does not. We will frequently write $r(\vec{x})$ instead of $r(x_1, \ldots, x_k)$ if the arity of the method does not matter.

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

Queries are represented by coded methods of the form

$$q \mathbin{@} \mathit{Number}, \ldots, \mathit{Number}\colon \mathit{Number}.$$

If $q\,(\vec{x})$ reduces to an object of class *Number*, then the tuple represented by $\vec{x}$ is considered part of the output; if $q\,(\vec{x})$ reduces to an object of class *Zero*, then the tuple represented by $\vec{x}$ is not considered part of the output.

8.3.2 Expressing PTIME Queries

We show that under the input/output conventions given above, method schemas can compute inflationary fixpoints of Datalog$^\neg$ programs (see [30] for a survey of these concepts). The results of Immerman [27] and Vardi [48] then imply that method schemas can compute all PTIME queries over finite ordered databases.

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

The Boolean functions can be coded as:

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} o' \ @\ \ & Number, \ldots, Number, \overbrace{Zero, \ldots, Zero}^{k} &&= \lambda\vec{x}\,\lambda\vec{n}.\, 0 \\ o' \ @\ \ & Number, \ldots, Number, \overbrace{Number, \ldots, Number}^{k} = \\ & \quad \lambda\vec{x}\,\lambda\vec{n}.\, Or\,(o'\,(\vec{x}, Pred_1^k\,(\vec{n}), \ldots, Pred_k^k\,(\vec{n})), \bar{e}\,(\vec{x})), \end{aligned}$$

where $\bar{e}$ is e with every occurrence of $o\,(\vec{x})$ replaced by $o'\,(\vec{x}, Pred_1^k\,(\vec{n}), \ldots, Pred_k^k\,(\vec{n}))$.

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

Theorem 8.3.1 *Under the input/output conventions listed above, MS with order can express every PTIME query.*

8.3.3 Simulating Method Schemas in Datalog

To prove the converse direction, we show that method schemas with order can be expressed by Datalog$^\neg$ programs with order. This, combined with the previous reduction, allows us to deduce that method schemas express exactly the PTIME queries.

We only sketch the reduction here. The details can be found in [25]. The idea is that a method p of arity k is simulated by means of a predicate P of arity $k+1$. For elements $x_1, \ldots, x_k, y$ of the domain, $P\,(x_1, \ldots, x_k, y)$ is true iff $p\,(x_1, \ldots, x_k) = y$. Base methods correspond to EDB predicates and coded methods correspond to IDB predicates. Classes are simulated by unary EDB predicates: for each class c there is a predicate Isa_c such that $\mathit{Isa}_c(x)$ is true iff object x is of class c. There is also a special interpreted predicate $\mathit{Pred}(x, y)$ which provides an ordering of the domain.

Coded method definitions are translated into Datalog rules as follows. We assume that all method definitions are explicit at all classes at which they are defined (i. e., no method is defined by inheritance). This might increase the size of the schema by a polynomial factor, but does not affect its data complexity. For each coded method definition $p\,@\,c_1, c_2, \ldots, c_k = \lambda x_1, \ldots, x_k.\, p_1(\ldots)$, there is a rule

$$\begin{aligned} P(X_1, X_2, \ldots, X_k, Y) :\!\!-\; & \mathit{Isa}_{c_1}(X_1), \ldots, \mathit{Isa}_{c_k}(X_k), \\ & P_{i_1}(\ldots), P_{i_2}(\ldots), \ldots, P_1(\ldots, Y); \end{aligned}$$

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

There are also additional rules that detect if an EDB predicate does not behave as expected. For example, there are rules that detect whether an object x is a member of two classes or of no classes (i. e., $\mathit{Isa}_c(x)$ is true for two different c or for no c), or whether a base method p is undefined or multiply defined for some set of arguments $x_1, \ldots, x_k$ (i. e., $P(x_1, \ldots, x_k, y)$ is true for two different y or for no y). If an inconsistency is detected, the computation is aborted by "flooding" the IDB predicates, i. e., deriving all possible facts within a single step (this is a standard technique for aborting Datalog computations, see, e. g., [23] for an example of its use).

It is easy to see that this encoding provides a faithful simulation of method schemas with order. Thus, together with the converse reduction established above, we have the following theorem:

Theorem 8.3.2 *Under the input/output conventions of Section 3.1, MS with order expresses exactly PTIME.*

Chapter 9

Conclusions and Open Questions

We have shown that the simply typed λ-calculus, without any added constructs, is a powerful functional language for computations over finite structures. Low-order fragments of the calculus augmented with an equality predicate are sufficient to express practically feasible queries. Furthermore, such fragments are sufficiently constrained to permit a detailed analysis of their expressibility, thereby providing a novel way of characterizing complexity classes. Another way of obtaining such characterizations is the framework of method schemas, which adds method resolution, late binding, and recursion to a strictly first-order calculus to obtain a functional language for object-oriented databases.

The research presented in this thesis leads to a number of interesting questions:

1. What is the expressive power of $\mathrm{TLI}_k^=$ and $\mathrm{MLI}_k^=$ for $k > 3$? In particular, is the converse of Theorem 6.3.5 true as conjectured in Section 7.5?

2. In our embeddings we use lists of tuples, which are inherently ordered. In [1, 11, 9] sets are used as basic constructs, and set iteration replaces list iteration in [11, 9]. Is it possible to eliminate the order dependence in our framework by augmenting the λ-calculus syntax and semantics with set iteration? One approach might also be to encode finite structures using a *generic* order computed by *graph canonization*.

3. 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 [12]?

4. We use ad-hoc reduction strategies to evaluate query terms efficiently. However, there is also work on general optimal reduction strategies (e. g., [36, 34]). How do such re-

duction strategies perform in our setting? In particular, can the bookkeeping required by these strategies be performed efficiently in the limited fragments of the λ-calculus we consider?

5. (Suggested by Pascal Van Hentenryck) It is well-known that outermost evaluation can be re-expressed in terms of innermost evaluation using CPS-like transformations. Would it be possible to reexpress all our expressibility results in a single evaluation strategy using such transformations?

6. There are various machine-independent characterizations of PTIME (e. g., [48, 27, 22, 29, 8, 35]). Is there a way of unifying some of these characterizations using the framework of the typed λ-calculus, maybe with suitable extensions?

Bibliography

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

[2] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings ACM SIGMOD*, pp. 159–173, 1989. (Also INRIA Technical Report 1022, 1989.)

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

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

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

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

[7] H. Barendregt. Functional Programming and Lambda Calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 321–363. Elsevier, 1990.

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

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

[10] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *Proceedings 4th ICDT*. Lecture Notes in Computer Science, Springer Verlag, 1992.

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

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

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

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

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

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

[17] B. Courcelle. Recursive Applicative Program Schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 459–492. Elsevier, 1990.

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

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

[20] 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.

[21] S. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification.* LNCS Vol. 36, Springer, 1975.

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

[23] G. Hillebrand, P. Kanellakis, H. Mairson, and M. Vardi. Tools for Datalog Boundedness. In *Proceedings 10th ACM PODS*, pp. 1–12, 1991.

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

[25] G. Hillebrand, P. Kanellakis, and S. Ramaswamy. Functional Programming Formalisms for OODB Methods. In *Proceedings of the NATO ASI Summer School on OODBs*, Turkey 1993. Lecture Notes in Computer Science, Springer Verlag, 1993.

[26] G. Hillebrand and P. Kanellakis. Functional Database Query Languages as Typed Lambda Calculi of Fixed Order. To appear in *Proceedings 13th ACM PODS*, 1994.

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

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

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

[30] P. Kanellakis. Elements of Relational Database Theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 1073–1156. Elsevier, 1990.

[31] P. Kanellakis, J. Mitchell, and H. Mairson. Unification and ML-type Reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pp. 444–479. MIT Press, 1991. (Also in *Proceedings 16th ACM POPL*, 1989, and *Proceedings 17th ACM POPL*, 1990.)

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

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

[34] J. Lamping. An Algorithm for Optimal Lambda Calculus Reduction. In *Proceedings 17th ACM POPL*, pp. 16–30, 1990.

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

[36] 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.

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

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

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

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

[41] R. Milner. The Standard ML Core Language. *Polymorphism*, **2** (1985), 28 pp. (An earlier version appeared in *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming.*)

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

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

[44] H. Schwichtenberg. An Upper Bound for Reduction Sequences in the Typed λ-Calculus. *Archive for Mathematical Logic*, **30** (1991), pp. 405–408.

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

[46] L. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Sci.*, **3** (1977), pp. 1–22.

[47] J. Ullman. *Principles of Database Systems*, 2nd ed. Computer Science Press, 1982.

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