# On the Expressive Power of Simply Typed and Let-Polymorphic Lambda Calculi

Gerd Hillebrand*
Fakultät für Informatik
Universität Karlsruhe
D-76137 Karlsruhe, Germany
ggh@ira.uka.de

Paris Kanellakis†
formerly of
Computer Science Department
Brown University
Providence, RI 02912

## Abstract

*We present a functional framework for descriptive computational complexity, in which the Regular, First-order, Ptime, Pspace, k-Exptime, k-Expspace ($k \geq 1$), and Elementary sets have syntactic characterizations. In this framework, typed lambda terms represent inputs and outputs as well as programs. The lambda calculi describing the above computational complexity classes are simply or* `let`*-polymorphically typed with functionalities of fixed order. They consist of: order 0 atomic constants, order 1 equality among these constants, variables, application, and abstraction. Increasing functionality order by one for these languages corresponds to increasing the computational complexity by one alternation. This exact correspondence is established using a* semantic evaluation *of languages for each fixed order, which is the primary technical contribution of this paper.*

## 1 Introduction

**Motivation:** The *simply typed λ-calculus* of Church [7] (*typed λ-calculus* or TLC) with its syntax and semantics is an essential part of most functional programming languages, e.g., ML, Miranda, Haskell, etc. A key feature of many such languages is `let`*-polymorphism* [31, 29]. TLC together with `let`-polymorphism is referred to as core-ML. *Constants of functionality order 0 and equality on these constants of order 1* are also part of all functional languages, e.g., `eq?` in Scheme. TLC and core-ML with constants and equality are TLC$^=$ and core-ML$^=$, respectively. *Functionality order* is a basic parameter in these languages, e.g., the common practice involves programming with low order functionalities. Given their essential roles, it is important to understand the expressive power of TLC$^{(=)}$ and core-ML$^{(=)}$ as well as that of their fixed order fragments.

**Related work in functional languages:** The standard approach to TLC expressibility considers computations over Church numerals (see, e.g., [3, 11, 35]). There are several results characterizing the expressive power of TLC over Church numerals. If inputs and outputs are Church numerals $\lambda s.\, \lambda z.\, s\,(s\,(s \ldots (s\, z)) \ldots)$ always typed as $\mathsf{Int}$ (where $\mathsf{Int} := (\tau \to \tau) \to \tau \to \tau$ for some type constant $\tau$ of order 0) Schwichtenberg [35] and Statman have shown that the expressible multi-argument functions of type $(\mathsf{Int}, \ldots, \mathsf{Int}) \to \mathsf{Int}$ (or equivalently, $\mathsf{Int} \to \cdots \to \mathsf{Int} \to \mathsf{Int}$) are exactly the *extended polynomials*. These are the functions generated by 0 and 1 using the operations addition, multiplication and conditional. The Schwichtenberg-Statman theorem exemplifies the *language uniform typed inputs convention*, where each input has one fixed simple type. This convention is the literature standard. It is used in [21] to code relational algebra, on finite structure inputs, in TLC$^=$ of order 3.

If inputs are Church numerals given more complex types than Int, exponentiation and predecessor can also be expressed [11]. (Statman, as quoted in [11], showed that equality, ordering, and subtraction are not expressible in TLC for any typing of Church numerals). The exponentiation examples of [11] illustrate the *program uniform typed inputs convention*. Namely, each program has a fixed simple type and its inputs must be typed uniformly with this program. However, the same input might have different types when considered as input to different programs. This convention is used in [21] to code fixpoint logic in TLC$^=$ of order 4 and higher-order logic in TLC$^=$.

*This work was done while visiting the University of Pennsylvania under DOE grant DE-FG02-94-ER-61923.

In fact other conventions are also possible. A most permissive one would be to type the outputs with fixed simple types (e.g., $\mathsf{Bool} := \tau \to \tau \to \tau$) and provide no typing restrictions on the inputs other than requiring Boolean outputs. Under this *nonuniform typed inputs convention* Statman's theorem that "deciding equivalence of normal forms of two simply typed $\lambda$-terms is not elementary recursive [37]" becomes an expressibility result. (The proof in [37] uses Meyer's theorem on the complexity of higher-order type theory [30]. For a simple proof of both see [28]). Under this convention it is possible to have inputs that are finite structures containing an equality predicate on the domain of each structure. This makes it possible to interpret Statman's and Meyer's theorems as expressibility of the Elementary sets in TLC [28] and to code Ptime in TLC of order 5 [21].

Simple types limit language uniform (but not program uniform or nonuniform) expressibility. This has provided motivation for examining more expressive typed calculi, such as the Girard-Reynolds second-order $\lambda$-calculus [12, 34] (polymorphism via type quantification) or Milner's ML [18, 31] (`let`-polymorphism and monomorphic fixpoints). For a recent survey see [32]. It is the consensus of programmers that simply typed data structures limit flexibility, whereas polymorphic ones are quite flexible. Although flexibility is hard to quantify, expressibility can be.

Here, the technical development is based on program uniformity, but has consequences on the language uniform expressibility of `let` polymorphism. The analysis highlights: (1) *the role of constants/equality* and (2) *ML's expressive power, even with limited polymorphism and no fixpoints.*

**Input typing conventions and type reconstruction:** Language uniformity and program uniformity naturally correspond to the Church and Curry views of the TLC. In the "Church view", a lambda term comes with a simple type. In the "Curry view", lambda terms come without simple types, which are reconstructed as needed. In studying expressibility one assumes the program is of fixed size, so both simple and `let`-polymorphic type reconstruction is trivially constant time. However, in terms of program size, fixed order `let`-polymorphic reconstruction is NP-hard; see the journal version of [20] (which corrects a mistaken polynomial claim in the proceedings version).

**Related work in descriptive computational complexity:** Some knowledge is assumed of the computational complexity classes Regular, First-order, Ptime (or P), NPtime (or NP), Pspace, $k$-Exptime (where 0-Exptime = Ptime, 1-Exptime = Exptime, 2-Exptime = doubly exponential time, etc.), $k$-Expspace, and Elementary and of the concept of ***alternation***; for a recent survey see [33]. The logical framework of first-order, higher-order, and fixpoint formulas over finite structures has been the principal vehicle of research in descriptive computational complexity and finite model theory. Starting with Fagin's characterization (in [10]) of NP by existential second-order formulas over finite structures, most computational complexity classes have been given such logical descriptions. For example, P has been characterized using fixpoint formulas [23, 40] over finite structures. The First-order sets have been characterized using first-order formulas over finite structures. These sets are a practically important class of relational database queries [9, 6]. The higher-order formulas have been extensively applied to the design of languages for complex-object databases; see the recent survey in [2] as well as [5, 24] for functional perspectives.

Functional formalisms have also been examined in the context of descriptive computational complexity. Since Cobham's early work there have been a number of interesting functional characterizations of P, e.g., [4, 8, 13, 17], not directly related to the TLC. A characterization related to the TLC was recently derived by Leivant and Marion [27]. In [27], the simply typed lambda calculus is augmented with a pairing operator and a "bottom tier" consisting of the free algebra of words over $\{0,1\}$ with associated constructor, destructor, and discriminator functions. With this addition, Leivant and Marion obtain various calculi which characterize P.

A research program was initiated in [21] and pursued in [19, 20, 1] to answer the natural question: "Is there a functional analogue of the logical framework of first-order, higher-order, and fixpoint formulas over finite structures?" This paper completes the research program. Its new contributions are explained in detail below and related to [21, 19, 20, 1]. Two important open questions remained, and are both answered here: (1) *understanding the expressive power of arbitrary functionality orders*, and (2) *understanding the expressive power of constants and equality.*

In terms of comparison of this paper's contribution with related work in descriptive computational complexity: (a) Higher-order logic has been used to express various exponential time and space classes, e.g., see [25, 22, 15, 26]. However, the exact characterizations given in this paper are more economical in basic primitives. (b) There are similarities between the approaches of [21, 20] and [27]. Both approaches add primitives to the TLC to encode P. The principal difference is that TLC$^{=}$ "equality and constants" appear to be one order weaker than the "bottom tier" of [27]. Thus, the resulting language for P (in [20]) has to use lambda terms that are one functional order higher than those in [27]. Although this might seem to be a "purer" functional characterization, it comes with a technical problem. Because of the increased order, evaluation of terms by $\beta$-reduction may require exponential time, even if only a Ptime computation is expressed. Therefore, one has to devise an alternate evaluation mechanism that always terminates in polynomial time. Addressing this technical issue (generalized to all orders) is the subject of this paper. The answer is based on the semantics of TLC [16, 38, 39] and *semantic evaluation* appropriate for constants and equality.

**Semantic evaluation vs. reduction evaluation:** In proving expressibility results for a fixed order language there are always two directions. First, a *lower bound on expressibility*, i.e., that each function of a computational complexity class is expressible. For functional languages this direction has been greatly simplified by Mairson's proof technique of *list iteration* [28], which allows an easy coding of the analogous proofs from predicate logic. The second direction is an *upper bound on expressibility*, i.e., that only functions of the computational complexity class are expressible. For predicate logic this direction is usually an immediate consequence of predicate logic semantics (e.g., for fixpoint formulas this direction corresponds to the statement that the Herbrand universe is finite and polynomial in the input size). For the functional languages examined here the analogous argument is nontrivial, because of the order of primitives that are freely combined in the languages. The primitives have functionality order low enough to encode the required sets (relations) but high enough to do so with exponentially many duplicate members (tuples). Semantic evaluation addresses duplicate elimination and other potential complications.

In functional programming languages evaluation is usually by some reduction strategy. However, expressing a Ptime function by program $p$ does not necessarily imply that $(p\,x)$ can be evaluated by a reduction strategy in polynomial time in input size $|x|$ . This is what the semantic evaluator guarantees, but not by reduction. Expressing a Ptime function should be contrasted to *efficiently embedding* it in a language, where also reduction evaluation is efficient. Such an embedding is given in [21] using a fixed increase of functionality order.

**Contributions and Overview:** A nontechnical contribution is the completion of a functional framework for descriptive computational complexity. The basic definitions are in Section 2. The expressibility of various languages is measured using their capabilities as set recognizers. There are a number of technical contributions:

1. (Section 3) Without equality or constants, it is possible to give a language uniform characterization of the Boolean functions and a program uniform characterization of the regular languages. The expressibility lower bounds use list iteration and the upper bounds semantic evaluation.

2. (Section 4) With equality and constants the functional framework is based on the input and output conventions first proposed in [21]. Here inputs are order 2 or higher terms encoding ordered finite structures and outputs are Booleans. At the minimum possible order 3, we have from [20] a language uniform characterization of the First-order queries in TLC$^=$, which is analogous to the Schwichtenberg-Statman language uniform characterization of the extended polynomials in order 3 TLC. For orders beyond 3, we generalize the program uniform characterizations of Ptime in order 4 TLC$^=$ and Pspace in order 5 TLC$^=$ from [20, 1] to program uniform characterizations of $k$-Exptime in order $2\,k + 4$ TLC$^=$ and $k$-Expspace in order $2\,k + 5$ TLC$^=$. This gives the full picture for the hyperexponential time and space complexity classes in TLC$^=$ and completes the research program started in [21].

   Our proofs involve a semantic evaluation technique, which is the main technical contribution of this paper. A simpler version was initially developed for the journal version of [20]. Here the argument is generalized to all orders, using a technique of Schwichtenberg from [36].

In Section 5, the framework developed for TLC$^{(=)}$ under the program uniform convention is applied to core-ML$^{(=)}$ under the language uniform convention. The final Section 6 is a discussion of the modifications that must be made to the framework to characterize NP (see [14]).

## 2 Definitions

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

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

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

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

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

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

Call a $\lambda$-term $E$ *well-typed* (or equivalently a term of TLC) and $\alpha$ a type of $E$, if $\Gamma \vdash E{:}\,\alpha$ is derivable by the above rules, for some $\Gamma$ and $\alpha$. An *explicitly typed* $\lambda$-term is a well-typed term $E$ together with a derivation of a type of $E$; we write such terms succinctly by providing type annotations on the free and bound variables of $E$.

The operational semantics of TLC are defined using *alpha and beta reduction*. See [3] for the definitions of reduction and substitution $F\,[x := E]$.

TLC$^=$ is obtained by enriching TLC with: (1) a type constant $\mathsf{o}$ different from $\tau$, (2) a countably infinite set $O = \{o_1, o_2, \ldots\}$ of expression constants of type $\mathsf{o}$, and (3) an expression constant $Eq$ of type $\mathsf{o} \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau \rightarrow \tau$. The type inference rules for TLC$^=$ are those of TLC augmented with axioms $o_i{:}\,\mathsf{o}$ $(i = 1, 2, \ldots)$ and $Eq{:}\,\mathsf{o} \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau \rightarrow \tau$. The reduction rules of TLC$^=$ are obtained by enriching the operational semantics of TLC as follows. For every pair of constants $o_i, o_j$, add to the TLC rules the reduction rule (known as a *delta-reduction* rule):

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

Terms of the TLC and TLC$^=$ are in *normal form*, when they cannot be further reduced by beta or delta reduction. See [3] for other reduction notions, not in TLC$^=$, such as $\eta$-reduction. From [3, 16] the following properties hold in TLC$^=$. (1) *Church-Rosser and Strong Normalization:* reductions lead to the same normal form and always terminate. (2) *Principal type property:* A simply typed term has a principal type, that is, a type from which all other types can be obtained via substitution of types for type variables. For semantic properties of TLC refer to [16, 38, 39].

**Core-ML and Core-ML$^=$:** For brevity we define `let`-polymorphism using simple type syntax (see [29]) with the extra inference rule (Let). The syntax of core-ML$^=$ (core-ML) is that of TLC$^=$ (TLC) augmented with one new term construct: `let` $\xi = \mathcal{E}$ `in` $\mathcal{E}$. Call a $\lambda$-term $E$ *well-typed* if $\Gamma \vdash E{:}\,\alpha$ is derivable by the inference rules (Var), (Abs), (App), and (Let) for some $\Gamma$ and $\alpha$.

$$(\text{Let}) \qquad \frac{\Gamma \vdash E{:}\,\alpha \quad \Gamma \vdash F\,[x := E]{:}\,\beta}{\Gamma \vdash \texttt{let}\ x = E\ \texttt{in}\ F{:}\,\beta}$$

The operational semantics is as for TLC$^=$ (TLC), where in addition `let` $x = E$ `in` $F$ is treated as $(\lambda x.\,F)\,E$. Also, core-ML$^=$ (core-ML) has all the above properties of TLC$^=$ (TLC), Church-Rosser etc. There is more flexibility in typing than in TLC$^=$ (TLC). For example, `let` $x = (\lambda z.\,z)$ `in` $(x\,x)$ is in core-ML but $(\lambda x.\,x\,x)(\lambda z.\,z)$ is not in TLC.

**Functionality Order:** The *order* of a type, which measures the higher-order functionality of a $\lambda$-term of that type, is defined as $\text{order}\,(\alpha) = 0$, if $\alpha$ is a type variable or type constant, and $\text{order}\,(\alpha) = \max(1 + \text{order}\,(\beta), \text{order}\,(\gamma))$, if $\alpha$ is an arrow type $\beta \rightarrow \gamma$. Refer to the *order of an explicitly typed $\lambda$-term* as the order of its type. The *order of a redex* $(\lambda x{:}\,\alpha.\,E)\,F$ occurring in an explicitly typed term is the order of $\alpha$. Finally, the above definitions and properties hold for fragments of TLC, TLC$^=$, core-ML, and core-ML$^=$, where the order of terms is bounded by some fixed $k$. In such fragments use the above inference rules but with all types restricted to order $k$ or less.

**Model Theory:** The model theory of TLC is standard (cf. [16]). All models used in this paper are finite and are full type frames over base type $\tau$ with binary domain $[\![\tau]\!] := \{0, 1\}$ and (in the TLC$^=$ case) base type $\mathsf{o}$ with varying, but finite domains $[\![\mathsf{o}]\!] \subset O$ depending on the query under consideration. The TLC$^=$ constants are interpreted in the natural way: $[\![o_i]\!] := o_i$ and $[\![Eq]\!] := \lambda x\,\lambda y\,\lambda u\,\lambda v.\ \textbf{if}\ x = y\ \textbf{then}\ u\ \textbf{else}\ v \in [\![\mathsf{o} \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau \rightarrow \tau]\!]$. The notation $[\![t]\!]$ denotes the value of a closed explicitly typed TLC or TLC$^=$ term $t$ in the model under consideration; here we assume that the type of $t$ does not contain type variables.

**List Iteration:** Let us briefly review how the basic technique of list iteration works. This technique is key to all lower bound proofs here and is quite powerful, see [28]. Let $\{E_1, E_2, \ldots, E_k\}$ be a set of $\lambda$-terms, each of type $\alpha$; then $L := \lambda c{:}\,\alpha \rightarrow \beta \rightarrow \beta.$

$\lambda n{:}\beta.\, c\, E_1\, (c\, E_2 \ldots (c\, E_{k-1}\, (c\, E_k\, n)) \ldots)$ is a $\lambda$-term of type $(\alpha \to \beta \to \beta) \to \beta \to \beta$. $L$ is called a *list iterator* encoding the list $E_1, E_2, \ldots, E_k$; the variables $c$ and $n$ abstract over list constructors *Cons* and *Nil*. For example, consider the standard encoding of Boolean logic, where *True* $:= \lambda x{:}\tau.\, \lambda y{:}\tau.\, x$ and *False* $:= \lambda x{:}\tau.\, \lambda y{:}\tau.\, y$, both of type $\mathsf{Bool} := \tau \to \tau \to \tau$ and the exclusive-or function can be written as *Xor* $:= \lambda p{:}\mathsf{Bool}.\, \lambda q{:}\mathsf{Bool}.\, \lambda x{:}\tau.\, \lambda y{:}\tau.\, p\, (q\, y\, x)\, (q\, x\, y)$. The parity function can be written as *Parity* $:= \lambda L{:}(\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool}.\, L\, \mathit{Xor}\, \mathit{False}$. Intuitively, when *Parity* is applied to a list iterator, its evaluation can be visualized as initializing an *accumulator* to *False* and then looping backwards through the elements in the list, setting the accumulator to the exclusive or of its previous value and the current list element at each stage.

**Inputs and Outputs:** For all languages in this paper, the output of each program is of fixed simple type $\mathsf{Bool} := \tau \to \tau \to \tau$. Moreover, programs and inputs are assumed to be closed terms, i.e., without free variables. It follows that each program, when applied to an input, returns one of the two normal forms *True* $:= \lambda x{:}\tau.\, \lambda y{:}\tau.\, x$ or *False* $:= \lambda x{:}\tau.\, \lambda y{:}\tau.\, y$.

In Section 3 dealing with the pure TLC, inputs are Booleans encoded as *True* or *False* and lists of Booleans, which are encoded as list iterators $\lambda c.\, \lambda n.\, c\, B_1\, (c\, B_2 \ldots (c\, B_k\, n) \ldots)$, where each $B_1, B_2, \ldots, B_k$ is either *True* or *False*. These iterators can be typed in many possible ways, e.g., $(\mathsf{Bool} \to \alpha \to \alpha) \to \alpha \to \alpha$ for each type $\alpha$. This is abbreviated as $\{\mathsf{Bool}\}^\alpha$, or, if the exact nature of $\alpha$ does not matter, as $\{\mathsf{Bool}\}^*$.

In Section 4 dealing with TLC$^=$, inputs are encodings of *ordered finite structures*. An ordered finite structure $(d, r_1, \ldots, r_m)$ of arity $(k_1, \ldots, k_m)$ consists of a finite set $d$ called its *domain*, relations $r_1, \ldots, r_m$ of arity $(k_1, \ldots, k_m)$ over $d$, and a linear ordering on $d$. For convenience, we assume that $d$ is a subset of the set of TLC$^=$ constants. We encode a linearly ordered domain $d = \{o_1, o_2, \ldots, o_N\}$ as the list iterator $\overline{d} := \lambda c.\, \lambda n.\, c\, o_1\, (c\, o_2 \ldots (c\, o_N\, n) \ldots)$ and a $k$-ary relation $r = \{(o_{1,1}, o_{1,2}, \ldots, o_{1,k}), \ldots, (o_{m,1}, o_{m,2}, \ldots, o_{m,k})\}$ as the list iterator

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

where each tuple of $r$ appears exactly once and the order of the tuples is lexicographical as determined by the order on $d$.

If $r$ contains at least two tuples, the principal type of $\overline{r}$ is $(\mathsf{o} \to \cdots \to \mathsf{o} \to \sigma \to \sigma) \to \sigma \to \sigma$, where $\sigma$ is an arbitrary type variable. The order of this type is 2, independent of the arity of $r$. We abbreviate this type as $\mathsf{o}_k^\sigma$. Instances of this type, obtained by substituting some type $\alpha$ for $\sigma$, are abbreviated as $\mathsf{o}_k^\alpha$, or, if the exact nature of $\alpha$ does not matter, as $\mathsf{o}_k^*$.

**Program Uniformity, REG, and TLI$_i^=$:** When characterizing regular languages in TLC and complexity classes beyond First-order in TLC$^=$, we adopt a *program uniform* typing convention, where each program term has fixed output type $\mathsf{Bool}$, but where different program terms may type their input encodings differently. We define languages REG and TLI$_i^=$, $i \geq 1$, as follows: REG is the set of all closed TLC terms explicitly typed with a type of the form $\{\mathsf{Bool}\}^* \to \mathsf{Bool}$, and TLI$_i^=$ is the set of all closed TLC$^=$ terms explicitly typed with a type of the form $\mathsf{o}^\alpha \to \mathsf{o}_{k_1}^{\alpha_1} \to \cdots \to \mathsf{o}_{k_m}^{\alpha_m} \to \mathsf{Bool}$, where $\alpha, \alpha_1, \ldots, \alpha_m$ have at most order $i$.

**Language Uniformity, BOOL, and TLI$_0^=$:** The program uniform convention should be contrasted with the *language uniform* convention, where all program terms type their inputs in the same way. This is the convention used in [11, 35]. We use it for characterizing the Boolean functions in TLC, the First-order queries over ordered finite structures in TLC$^=$, and for all characterizations in ML$^=$. We define languages BOOL and TLI$_0^=$ as follows: Let BOOL be the set of all closed TLC terms explicitly typed as $\mathsf{Bool} \to \cdots \to \mathsf{Bool} \to \mathsf{Bool}$, and let TLI$_0^=$ be the set of all closed TLC$^=$ terms explicitly typed as $\mathsf{o}^\tau \to \mathsf{o}_{k_1}^\tau \to \cdots \to \mathsf{o}_{k_l}^\tau \to \mathsf{Bool}$, where $l \geq 0$.

We can define analogous languages MLREG, MLBOOL, and MLI$_i^=$, $i \geq 0$ using ML and ML$^=$ terms instead of TLC and TLC$^=$ terms. These differ in one important aspect from their TLC and TLC$^=$ counterparts: The use of polymorphism turns program uniformity into language uniformity. This can be shown using Milner's polymorphic notation, but is not included here to keep the presentation short.

## 3 On the Expressive Power of TLC

Our first expressibility result in the pure typed $\lambda$-calculus is a simple characterization of the Boolean functions.

**Theorem 3.1** *The functions expressible by terms in BOOL are exactly the Boolean functions.*

**Proof:** Any closed TLC term of type $\mathsf{Bool} \to \mathsf{Bool} \to \cdots \to \mathsf{Bool} \to \mathsf{Bool}$ reduces, once its arguments have been supplied, to a closed normal form of type Bool, hence either to *True* or *False*, and therefore defines a Boolean function.

Conversely, any Boolean function can be translated into a TLC term by writing it as Boolean formula and translating the formula into TLC using the connectives $And := \lambda p\colon \mathsf{Bool}.\, \lambda q\colon \mathsf{Bool}.\, \lambda u\colon \tau.\, \lambda v\colon \tau.\, p\,(q\,u\,v)\,v$, $Or := \lambda p\colon \mathsf{Bool}.\, \lambda q\colon \mathsf{Bool}.\, \lambda u\colon \tau.\, \lambda v\colon \tau.\, p\,u\,(q\,u\,v)$, and $Not := \lambda p\colon \mathsf{Bool}.\, \lambda u\colon \tau.\, \lambda v\colon \tau.\, p\,v\,u$. □

**Remark 3.2** The theorem remains valid if we allow more liberal typings of the form $\mathsf{Bool}^{\alpha_1} \to \mathsf{Bool}^{\alpha_2} \cdots \to \mathsf{Bool}^{\alpha_k} \to \mathsf{Bool}$, where $\mathsf{Bool}^{\alpha_i} := \alpha_i \to \alpha_i \to \alpha_i$ and $\alpha_i$ is an arbitrary type. For any closed TLC term $Q$ of such type there is a closed TLC term $Q'$ of type $\mathsf{Bool} \to \mathsf{Bool} \to \cdots \to \mathsf{Bool} \to \mathsf{Bool}$ that computes the same Boolean function. $Q'$ can be defined as $\lambda p_1\colon \mathsf{Bool} \ldots \lambda p_k\colon \mathsf{Bool}.\, Q\,\tilde{p}_1 \ldots \tilde{p}_k$, where $\tilde{p}_i := \lambda u\colon \alpha_i.\, \lambda v\colon \alpha_i.\, \lambda \vec{x}\colon \vec{\beta}.\, p\,(u\,\vec{x})\,(v\,\vec{x})$ and $\vec{x}\colon \vec{\beta}$ is chosen so that $u\,\vec{x}$ has type $\tau$.

**Remark 3.3** If a Boolean function $f$ is given by a circuit (rather than a formula), a TLC term computing $f$ can be found whose size is linear in the size of the circuit. This is done by using $\lambda$-abstraction to simulate shared subexpressions: if expression $E$ occurs multiple times in expression $F$, we can replace each occurrence with a variable $x$ and then write $(\lambda x.\, F)\,E$ instead of the original $F$.

More interestingly, there is also a simple characterization of the regular languages over the alphabet $\Sigma := \{0, 1\}$ (or, with the obvious modifications, any finite alphabet). We encode a word $w = b_1 b_2 \ldots b_m$ with $b_i \in \{0, 1\}$ as a list iterator $\overline{w} := \lambda c.\, \lambda n.\, c\, B_1\, (c\, B_2 \ldots (c\, B_{m-1}\, (c\, B_m\, n)) \ldots)$, where $B_i := \mathit{False}$ if $b_i = 0$ and $B_i := \mathit{True}$ if $b_i = 1$. Under this encoding, any closed TLC term of type $\{\mathsf{Bool}\}^* \to \mathsf{Bool}$ defines a Boolean function on words over $\Sigma$.

**Theorem 3.4** *The functions expressible by terms in REG are exactly the characteristic functions of regular languages over the alphabet* $\{0, 1\}$.

**Proof:** If $L$ is a regular language over $\{0, 1\}$, a $\lambda$-term $Q$ recognizing $L$ can be constructed as follows. First, construct a deterministic finite automaton $M$ recognizing the *reversal* of $L$, i.e., the set of bit strings $\{b_m \ldots b_2\, b_1 \mid b_1\, b_2 \ldots b_m \in L\}$ (which is also regular). Let $\{s_1, \ldots, s_n\}$ be the set of states of $M$, where $s_1$ is the starting state and $\{s_{m+1}, \ldots, s_n\}$ are the accepting states. For $1 \le i \le n$, let $s_{t_i}$ and $s_{f_i}$ be the states that $M$ enters from state $s_i$ when reading a 1 or 0 bit, respectively. We encode each state $s_i$ as a projection function $\pi_i := \lambda x_1\colon \tau \ldots \lambda x_n\colon \tau.\, x_i$ of type $\chi := \tau \to \tau \to \cdots \to \tau$, and we simulate a run of $M$ on the reversal of a string $w$ by a list iteration over the list representation of $w$, where we keep the current state in the "accumulator" (beginning with $\pi_1$) and update it as the iteration proceeds from the last to the first bit in $w$. The result of the iteration encodes the state of $M$ after reading the reversal of $w$, and we return *True* or *False* depending on whether the state is accepting or not. The complete term $Q$ is given by:

$$
\begin{aligned}
&Q\colon \{\mathsf{Bool}\}^{\chi} \to \mathsf{Bool} := \\
&\quad \lambda w\colon \{\mathsf{Bool}\}^{\chi}.\, \lambda u\colon \tau.\, \lambda v\colon \tau. \\
&\qquad w\ (\lambda b\colon \mathsf{Bool}.\, \lambda s\colon \chi.\, \lambda x_1\colon \tau \ldots \lambda x_n\colon \tau. \\
&\qquad\qquad b\,(s\, x_{t_1} \ldots x_{t_n})\,(s\, x_{f_1} \ldots x_{f_n})) \\
&\qquad \pi_1 \\
&\qquad \underbrace{v\,v \ldots v}_{m}\underbrace{u\,u \ldots u}_{n-m}
\end{aligned}
$$

Conversely, let $Q$ be a closed TLC term of type $\{\mathsf{Bool}\}^{\alpha} \to \mathsf{Bool}$ for some type $\alpha$ (we may assume that $\alpha$ does not contain type variables) and let $L$ be the set of bit strings whose encodings are accepted by $Q$. We will construct a finite automaton $M$ recognizing $L$. To this end, consider the semantics of TLC in the full type frame (cf. [16]) over base domain $[\![\tau]\!] := \{0, 1\}$. The state space of $M$ will be the (finite) function space $S := [\![\{\mathsf{Bool}\}^{\alpha}]\!]$ and the initial state will be $s_0 := [\![\lambda c\colon \mathsf{Bool} \to \alpha \to \alpha.\, \lambda n\colon \alpha.\, n]\!]$. For every bit string $w$, $M$ has a transition labeled 0 from $[\![\overline{w}]\!]$ to $[\![\overline{w0}]\!]$ and a transition labeled 1 from $[\![\overline{w}]\!]$ to $[\![\overline{w1}]\!]$. A state $s$ is accepting iff $[\![Q]\!](s) = [\![\mathit{True}]\!]$. It is easy to see that $M$ is deterministic (if $[\![\overline{w}]\!] = [\![\overline{v}]\!]$ then $[\![\overline{w1}]\!] = [\![\lambda c.\, \lambda n.\, \overline{w}\, c\, (c\, \mathit{True}\, n)]\!] = \lambda c.\, \lambda n.\, [\![\overline{w}]\!]\, (c, c([\![\mathit{True}]\!], n)) = \lambda c.\, \lambda n.\, [\![\overline{v}]\!]\, (c, c([\![\mathit{True}]\!], n)) = [\![\overline{v1}]\!]$) and that after reading a bit string $w$, $M$ will be in state $[\![\overline{w}]\!]$. Hence, $M$ accepts $w$ if and only if $[\![Q]\!]\,([\![\overline{w}]\!]) = [\![Q\, \overline{w}]\!] = [\![\mathit{True}]\!]$, i. e., $(Q\, \overline{w})$ reduces to *True*. □

**Remark 3.5** The proof above shows that a term in REG with the simplest possible type $\{\mathsf{Bool}\} \to \mathsf{Bool}$ can be simulated by a DFA with $|[\![\{\mathsf{Bool}\}]\!]| = 2^{2^{33}}$ states. However, most of these states are not of the form $[\![\overline{w}]\!]$ with $w \in \{0, 1\}^*$ and therefore have no transitions associated with them, so they may be deleted from the automaton. Moreover, some of the remaining states are $\lambda$-indistinguishable in the sense that there is no $\lambda$-definable function of type $[\![\{\mathsf{Bool}\}]\!] \to [\![\mathsf{Bool}]\!]$ that can tell them apart. A careful case analysis shows that in fact only five states are distinguishable: these correspond to the empty string, strings containing only 1's, strings containing only 0's, strings containing both

1's and 0's beginning with a 0, and strings containing both 1's and 0's beginning with a 1, respectively. It follows that the language recognized by any $\lambda$-term of type $\{\mathsf{Bool}\} \rightarrow \mathsf{Bool}$ is the union of some of these five basic languages. In particular, any such language can also be recognized by a uniform family of constant-depth, unbounded fan-in circuits.

## 4 On the Expressive Power of TLC$^=$

In this section, we generalize the results of [20, 1] to provide characterizations in TLC$^=$ of the $k$-Exptime and $k$-Expspace complexity classes for all $k \geq 0$. In keeping with these earlier papers, we consider computations over ordered finite structures, which are encoded in TLC$^=$ as described in Section 2. Equivalently, we could have considered computations over bit strings encoded as lists of (*position*, *value*) pairs, where the *position* field contains a unique constant for each bit and *value* is either *True* or *False*.[1]

**Theorem 4.1** *Under the finite structure input/output convention described in Section 2: (1) TLI$_0^=$ expresses language uniformly exactly the first-order queries over ordered finite structures; and (2) for $k \geq 0$, TLI$_{2k+1}^=$ and TLI$_{2k+2}^=$ express program uniformly the k-Exptime and k-Expspace queries over ordered finite structures, respectively.*

The cases TLI$_0^=$, TLI$_1^=$, and TLI$_2^=$ were first proven in [20, 1] for queries producing relations (instead of Booleans) as output. Since a Boolean query can be easily modified to produce a relation instead (representing *True* as a nonempty and *False* as an empty relation), they also apply in our current Boolean query setting. The rest of this section is devoted to proving the cases TLI$_k^=$ for $k \geq 3$.

### 4.1 Lower Bounds on Expressibility

We briefly sketch how $k$-Exptime and $k$-Expspace computations can be simulated in TLC$^=$. The technique is an adaptation of [28] and builds on the encodings of first-order and Ptime queries given in [21].

To simulate an arbitrary $k$-Exptime ($k$-Expspace) query $Q$ over an ordered finite structure $\mathcal{S}$, it suffices to iterate a first-order query $Q'$ a $k$-hyperexponential (($k$ + 1)-hyperexponential) number of times over a structure $\mathcal{S}'$ of size $k$-hyperexponential in the size of $\mathcal{S}$: namely, pick $Q'$ as describing the transition function of a Turing machine computing $Q$ and $\mathcal{S}'$ as describing a tape of $k$-hyperexponential length with an encoding of $\mathcal{S}$ in its first few cells.

[1] The results of the previous section show that the *position* component is essential if one wants to go beyond regular languages.

There are two crucial steps in coding this iteration. The first is the construction of a Church numeral *Crank* of $k$-hyperexponential or $(k+1)$-hyperexponential size, which will serve to iterate the query $Q'$ the required number of times. This is done, e. g., by computing the length of each input relation (as a Church numeral), computing a suitable product of these lengths, and finally exponentiating this product $k$ or $k+1$ times. The exponentiation requires that each input relation is typed with an order $k+2$ or $k+3$ type rather than its order 2 principal type.

The second step is the construction of the domain of $\mathcal{S}'$. Here we use the model theory of TLC. Let $d = \{o_1, \ldots, o_N\}$ be the domain of $\mathcal{S}$. Let $\alpha_1$ be the type $\mathsf{o} \rightarrow \mathsf{o} \rightarrow \cdots \rightarrow \mathsf{o} \rightarrow \mathsf{Bool}$ (the number of occurrences of $\mathsf{o}$ is chosen depending on the exact complexity of $Q$), and for $1 < i \leq k$, let $\alpha_i := \alpha_{i-1} \rightarrow \alpha_{i-1}$. Let $D_i = [\![\alpha_i]\!]$ be the domain of $\alpha_i$ in the full type frame over base domains $[\![\tau]\!] := \{0, 1\}$ and $[\![\mathsf{o}]\!] := d$. Note that $|D_k|$ is $k$-hyperexponential in $|d|$. Each $D_i$ may be ordered in a natural way, based on the ordering of $d$.

For $1 \leq i \leq k$, it is possible to write TLC$^=$ terms $Equal_i\colon \alpha_i \rightarrow \alpha_i \rightarrow \mathsf{Bool}$, $LessThan_i\colon \alpha_i \rightarrow \alpha_i \rightarrow \mathsf{Bool}$, $Min_i\colon \alpha_i$, $Max_i\colon \alpha_i$, and $Succ\colon \alpha_i \rightarrow \alpha_i$ coding equality, ordering, minimal and maximal elements, and the successor function on $D_i$, respectively. Moreover, one can code a quantifier operator $Exists_i\colon (\alpha_i \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Bool}$ such that for any closed term $P\colon \alpha_i \rightarrow \mathsf{Bool}$, $(Exists_i\, P) \rhd True$ iff there exists a value $[\![v]\!] \in [\![\alpha_i]\!]$ with $(P\, v) \rhd True$. This operator essentially uses the term *Crank* constructed above to build a large disjunction $Or\,(P\, Min_i)\,(Or\,(P\,(Succ_i\, Min_i))\,(Or\,(P\,(Succ_i\,(Succ_i\, Min_i)))\ldots))$. Finally, one can write "lifting" operators that transform the input relations, which are defined on $d$, into relations on $D_k$. The coding of all these operators is complicated to some extent by the restrictions of the monomorphic type system of TLC; see the "type laundering" technique of [21] for an illustration of these difficulties.

Once $D_k$ and its associated predicates have been constructed, a Turing machine computing the original query $Q$ can be coded by expressing its transition function as a first-order query $Q'$ over domain $D_k$ and using *Crank* to iterate it. Since the values passed from one stage of the iteration to the next are predicates over $D_k$ and require order $k+1$ types to be represented, the order of the input relations must be increased by an additional $k+1$ levels above the order $k+2$ or $k+3$ imposed during the construction of *Crank*. Thus, the total order of the input relations becomes $2\,k+3$ for a simulation of $k$-Exptime and $2k+4$ for $k$-Expspace, which makes the corresponding TLC$^=$ terms members of TLI$^=_{2k+1}$ and TLI$^=_{2k+2}$, respectively.

## 4.2 Upper Bounds on Expressibility

Our goal in this section is to prove the second half of Theorem 4.1, namely:

**Theorem 4.2** *Queries expressed by query terms in TLI$^=_{2k+1}$ and TLI$^=_{2k+2}$ can be evaluated in k-Exptime and k-Expspace, respectively.*

It is easy to see that these bounds cannot be obtained by a purely syntactic evaluation mechanism (that is, $\beta\eta$-reduction), because TLC$^=$ terms of order $\approx 2k$ may have normal forms of up to $2k$-hyperexponential size. We use therefore again a semantic evaluation technique. Due to the presence of constants $\{o_1, o_2, \ldots\}$, however, we are now dealing with an infinite base domain and therefore cannot precompute the semantics of a query term. Instead, for any given input, we have to compute the semantics of the query term and input together in a model that is based on the (finite) set of constants appearing in the query term and the input.

In the following, let $Q$ be a TLI$^=_k$ query term ($k \geq 1$). We assume that the explicit type of $Q$ does not contain type variables. Let $\overline{d}, \overline{r_1}, \ldots, \overline{r_l}$ be a legal input for $Q$ (i.e., an encoding of a finite structure $(d, r_1, \ldots, r_l)$, where the arities of $r_1, \ldots, r_l$ are the ones stipulated by the type of $Q$), and let $D$ be the set of constants occurring in $Q$ or $d$. We will compute $[\![Q\,\overline{d}\,\overline{r_1}\ldots\overline{r_l}]\!]$ in the full type frame over base domains $[\![\mathsf{o}]\!] := D$ and $[\![\tau]\!] := \{0,1\}$, where $[\![o_i]\!] := o_i$ and $[\![Eq]\!] := \lambda x\,y\,u\,v.\,\mathbf{if}\ x = y\ \mathbf{then}\ u\ \mathbf{else}\ v$. Since the result must be either $[\![True]\!]$ or $[\![False]\!]$ and these two values are different elements of $[\![\mathsf{Bool}]\!]$, the result will tell us the outcome of the query.

During the computation, we store the value $[\![E]\!]$ of a closed expression $E{:}\,\alpha$ as a table containing the graph of $[\![E]\!]$. That is, if $\alpha \equiv \beta_1 \to \cdots \to \beta_n \to \gamma$ with $\gamma \equiv \mathsf{o}$ or $\gamma \equiv \tau$, then $[\![E]\!]$ is stored as $\{(v_1, \ldots, v_n, w) \mid v_i \in [\![\beta_i]\!],\ w \in [\![\gamma]\!],\ w = [\![E]\!](v_1, \ldots, v_n)\}$. For expressions of order $m$, the size of this table is (m-1)-hyperexponential in the size of $D$.

We are going to define a family of evaluation procedures $Eval_k$, where $k = 0, 1, 2, \ldots$. For a closed explicitly typed TLC$^=$ term $E$, each $Eval_k(E)$ computes $[\![E]\!]$, but the evaluators differ in the reduction strategy they use. Roughly speaking, $Eval_k$ uses "call-by-value" for redexes up to order $k$ and "call-by-name" (i.e., $\beta$-reduction) for redexes of higher order. Thus, $Eval_0$ uses pure $\beta$-reduction, which requires very deep recursion, but very little storage for intermediate results, whereas higher order evaluators use a more shallow recursion, but larger intermediate results. It turns out that the best tradeoff is obtained when $k$ is about half as large as the maximum of the orders of the subterms of $E$, which leads to the time and space bounds in the theorem above.

To simplify the presentation, we adopt the following conventions: (1) In any given term, all bound variables are distinct from each other and all free variables, and (2) all terms are fully $\eta$-expanded in the sense that any subterm of non-zero order is either an abstraction or occurs as the left part of an application (this can always be achieved by suitable $\eta$-expansion). Moreover, we allow values as part of $\lambda$-terms; more precisely, we introduce for every type $\alpha$ and every value $v \in [\![\alpha]\!]$ a constant $c_v{:}\,\alpha$ with $[\![c_v]\!] := v$, which we identify with $v$.

Under these conventions, a closed explicitly typed TLC$^=$ term $C$ can take one of the following three forms: (1) $C \equiv v\,E_1 \ldots E_n$, where $n \geq 0$ and $v$ is a value; (2) $C \equiv (\lambda x_1{:}\,\alpha_1 \ldots \lambda x_n{:}\,\alpha_n.\,E)\,F_1 \ldots F_n$, where $n \geq 1$; and (3) $C \equiv \lambda x_1{:}\,\alpha_1 \ldots \lambda x_n{:}\,\alpha_n.\,E$, where $n \geq 1$. For these three cases, $Eval_k(C)$ is defined as follows.

$$
\begin{aligned}
&Eval_k(v\,E_1 \ldots E_n) := \\
&\quad Apply(v, Eval(E_1), \ldots, Eval(E_n)) \\
&Eval_k((\lambda x_1{:}\,\alpha_1 \ldots \lambda x_n{:}\,\alpha_n.\,E)\,F_1 \ldots F_n) := \\
&\quad Eval_k(E\,[x_1 := G_1, \ldots, x_n := G_n]), \\
&\quad \text{where } G_i := Eval_k(F_i) \text{ if order}(\alpha_i) \leq k \\
&\quad \text{and } G_i := F_i \text{ if order}(\alpha_i) > k \\
&Eval_k(\lambda x_1{:}\,\alpha_1 \ldots \lambda x_n{:}\,\alpha_n.\,E) := \\
&\quad \{(v_1, \ldots, v_n, w) \mid v_i \in [\![\alpha_i]\!],\ w = Eval_k(E\,[\vec{x} := \vec{v}])\}
\end{aligned}
$$

Here, *Apply* denotes function application in the type frame. Since we represent all functions as tables, *Apply* amounts to a table lookup.

**Lemma 4.3** *For any closed TLC$^=$ term C, $Eval_k(C)$ yields (the graph of) $[\![C]\!]$.*

**Proof:** By a straightforward induction on the structure of $C$. □

Let $d_k(C)$ be the depth of the recursion tree generated by the call $Eval_k(C)$ (including the initial call) and let $d(C)$ be the depth of the syntax tree of $C$. We use a technique of Schwichtenberg [36] to obtain a bound on $d_k(C)$ in terms of $d(C)$.

**Lemma 4.4** *Let M be a TLC$^=$ term with free variables among $\{y_1, \ldots, y_l\}$, let $C_1, \ldots, C_l$ be closed TLC$^=$*

*terms with* $\mathrm{type}(C_i) = \mathrm{type}(y_i)$ *and* $\mathrm{order}(C_i) \leq k+1$ *for* $1 \leq i \leq l$*, let* $\rho$ *be the substitution* $[y_1 := C_1, \ldots, y_l := C_l]$*, let* $\sigma$ *be the substitution* $[y_1 := [\![C_1]\!], \ldots, y_l := [\![C_l]\!]]$*, and let* $m := \max_{1 \leq i \leq l} d_k(C_i)$*. Then* $d_k(M\rho) \leq d_k(M\sigma) + m$*.*

**Proof:** (Sketch) The evaluation of $M\rho$ mimicks that of $M\sigma$, except when a subterm $(C_i\, E_1 \ldots E_n)$ is encountered. In that case, $M\sigma$ will contain the subterm $([\![C_i]\!]\, E_1 \ldots E_n)$ instead. After evaluating the arguments $E_1, \ldots, E_n$, the evaluator of $M\sigma$ can compute the value of the subterm by a simple table lookup, but the evaluator of $M\rho$ must instead evaluate a copy of $C_i$ with $[\![E_1]\!], \ldots, [\![E_n]\!]$ for arguments. Thus, the recursion tree for $M\rho$ may be as tall as the recursion trees for $M\sigma$ and $C_i$ stacked on top of each other. Note that the condition $\mathrm{order}(C_i) \leq k+1$ is essential: otherwise, $Eval_k$ would call some of the $E_i$ by name and the recursion tree for $(C_i\, E_1 \ldots E_n)$ would be deeper than $d_k(C_i)$. $\square$

**Lemma 4.5** *Let* $C$ *be closed and* $k \geq 0$*. Then* $d_k(C) \leq 2^{d_{k+1}(C)} - 1$*.*

**Proof:** (Sketch) The recursion trees of $Eval_{k+1}(C)$ and $Eval_k(C)$ differ in those places where a redex of order $k+1$ is evaluated. In that case, $Eval_{k+1}$ uses call by value, whereas $Eval_k$ uses call by name. According to the previous lemma, this may at most double the depth of the recursion tree. There may be at most $d_{k+1}(C)$ such redexes along any path in the recursion tree of $Eval_{k+1}(C)$, so $d_k(C)$ is at most exponential in $d_{k+1}(C)$. $\square$

**Lemma 4.6** *Let* $C$ *be closed and* $k$ *be the maximum order of any redex in* $C$*. Then* $d_k(C) \leq d(C)$*.*

**Proof:** (Sketch) Since all redexes in $C$ are of order $\leq k$, $Eval_k(C)$ will never perform any $\beta$-reductions. It follows that the argument of every recursive call of $Eval_k$ has a strictly smaller depth than that of the parent call, hence $d_k(C) \leq d(C)$. $\square$

**Lemma 4.7** *Let* $Q$ *be a* $TLI_k^=$ *query term and let* $\overline{d}, \overline{r_1}, \ldots, \overline{r_l}$ *be a legal input for* $Q$*. Let* $Q'$ *be the term obtained from* $(Q\,\overline{d}\,\overline{r_1} \ldots \overline{r_l})$ *by contracting all redexes of order* $\geq k$*. Then* $d(Q')$ *is polynomial in* $|d|$*. Furthermore,* $Q'$ *can be computed in exponential time.*

**Proof:** By an analysis of the structure of $Q$. $\square$

**Lemma 4.8** *Let* $Q$ *be a* $TLI_k^=$ *query term for* $k \geq 3$*,* $\overline{d}, \overline{r_1}, \ldots, \overline{r_l}$ *be a legal input for* $Q$*, and* $Q'$ *be defined as above. Then: (1) if* $k = 2m+1$*,* $Eval_{m+1}(Q')$ *will produce* $[\![(Q\,\overline{d}\,\overline{r_1} \ldots \overline{r_l})]\!]$ *in* $m$*-Exptime, and (2) if* $k = 2m+2$*,* $Eval_{m+1}(Q')$ *will produce* $[\![(Q\,\overline{d}\,\overline{r_1} \ldots \overline{r_l})]\!]$ *in* $m$*-Expspace.*

**Proof:** (Sketch) First observe that $Eval_{m+1}(Q') = [\![Q']\!] = [\![(Q\,\overline{d}\,\overline{r_1} \ldots \overline{r_l})]\!]$, so the evaluation produces the correct result. Second, since $Q'$ has no redexes of order $\geq k$, $d_{k-1}(Q')$ is bounded by $d(Q')$ and hence polynomial in $|d|$. It follows that $d_{m+1}(Q')$ is at most $(k-m-2)$-hyperexponential in $|d|$.

Consider the time and space consumed at each node in the recursion tree of $Eval_{m+1}(Q')$. It is easy to see that $Eval_{m+1}$ invokes itself only on terms of order $m+1$ or less, so the largest values that $Eval_{m+1}$ has to manipulate are of order $m+1$. These values are stored represented as tables of size at most $m$-hyperexponential in $|d|$. The time and space consumed at each node are bounded by the size of these values[2], and so is the number of children of each node.

It follows that if $k = 2m+1$ is odd, the recursion tree of $Eval_{m+1}(Q')$ is of at most $(k-m-2 = m-1)$-hyperexponential depth with a fanout at most $m$-hyperexpontial, so the total number of nodes is at most $m$-hyperexponential. Each node consumes at most $m$-hyperexponential time, so the total computation can be carried out in $m$-Exptime.

Similarly, if $k = 2m+2$ is even, the recursion tree of $Eval_{m+1}(Q')$ is of at most $(k-m-2 = m)$-hyperexponential depth with each node consuming at most $m$-hyperexponential space, so the total computation can be carried out in $m$-Expspace. $\square$

Thus, the complete algorithm for evaluating $\mathrm{TLI}_k^=$ queries, where $k \geq 3$, is as follows: First, compute the set of constants appearing in the input and the query term to determine $[\![o]\!]$, then compute $Q'$ as defined above, and finally compute $Eval_{\lfloor (k+1)/2 \rfloor}(Q')$. Together with the Ptime and Pspace bounds proven for $\mathrm{TLI}_1^=$ and $\mathrm{TLI}_2^=$ queries in [19], this establishes Theorem 4.2.

## 5 On the Expressive Power of core-ML and core-ML$^=$

The theory of `let`-polymorphism can be explained using either quantified type variables [31] or using simple types and the rule of Section 2 [29]. The approaches are essentially equivalent. What can be shown using the quantified type variable notation, is that the above program uniform expressibility results become language uniform expressibility results for MLREG and $\mathrm{MLI}_i^=$:

[2]We assume here that $\lambda$-terms are represented as dags and substitution is implemented as a constant-time operation, e. g., by maintaining environments.

**Theorem 5.1** *MLREG expresses language uniformly exactly the regular languages. For $k \geq 0$, $MLI^{=}_{2k+1}$ and $MLI^{=}_{2k+2}$ express language uniformly exactly the $k$-Exptime and $k$-Expspace queries over ordered finite structures, respectively.*

## 6 Discussion

The functional framework for descriptive computational complexity presented here has syntactic characterizations for the Regular, First-order, Ptime, Pspace, $k$-Exptime, $k$-Expspace ($k \geq 1$), and Elementary sets. These characterizations highlight the symmetries between time and space that are also present in the theory of alternation. The Ptime vs Pspace question becomes a question of expressibility of order 1 vs. order 2.

The framework does not include a syntactic characterization for NP. This is not surprising, given the deterministic nature of computations modeled by lambda calculi. Also, the syntactic classes characterized are all closed under complement, and this makes a similar NP characterization unlikely. There is a simple (and conventional) way to capture NP. Add one primitive to the lambda calculus, namely a *coin* primitive that converts to both booleans. Coins can be part of the input list-iterators. We have not pursued this approach here, since it would result in violating Church-Rosser, even while preserving termination.

**A personal remark.** Paris Kanellakis died in a plane crash on December 20, 1995, a few days after we finished this work. I had the privilege of enjoying his guidance, company, and friendship during my Ph.D. studies and our collaboration afterwards. With sadness, but also with warm remembrance, I dedicate this paper to his memory.

## References

[1] S. Abiteboul and G. Hillebrand. Space Usage in Functional Query Languages. In *Proceedings of the 5th International Conference on Database Theory*, pp. 439–454. Springer LNCS 893, 1995.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

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

[5] P. Buneman, S. Naqvi, V. Breazu-Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, **149** (1995), pp. 3–49.

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

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

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

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

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

[11] 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.

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

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

[14] E. Grädel and Y. Gurevich. Tailoring Recursion for Complexity. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, pp. 106–116. Springer LNCS 820, 1994.

[15] S. Grumbach and V. Vianu. Tractable Query Languages for Complex Object Databases. In *Proceedings of the 10th ACM Symposium on the Principles of Database Systems*, pp. 315–327. ACM Press, 1991.

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

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

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

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

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

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

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

[23] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Computation*, **68** (1986), pp. 86–104.

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

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

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

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

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

[29] H. Mairson. Quantifier Elimination and Parametric Polymorphism in Programming Languages. *J. Functional Programming*, **2** (1992), pp. 213–226.

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

[31] R. Milner. A Theory of Type Polymorphism in Programming. *J. Computer and System Sciences*, **17** (1978), pp. 348–375.

[32] J. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science, Volume B*, pp. 365–458. Elsevier, 1990.

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

[34] J. Reynolds. Towards a Theory of Type Structure. In *Proceedings of the Paris Colloquium on Programming*, pp. 408–425. Springer LNCS 19, 1974.

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

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

[37] R. Statman. The Typed $\lambda$-Calculus is Not Elementary Recursive. *Theoretical Computer Science*, **9** (1979), pp. 73–81.

[38] R. Statman. Completeness, Invariance, and $\lambda$-Definability. *J. Symbolic Logic*, **47** (1982), pp. 17–26.

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

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