An Analysis of the Core-ML Language: Expressive Power and Type Reconstruction
(invited paper)

Paris C. Kanellakis*, Gerd G. Hillebrand**, Harry G. Mairson***

Abstract. Core-ML is a basic subset of most functional programming languages. It consists of the simply typed (or monomorphic) λ-calculus, simply typed equality over atomic constants, and `let` as the only polymorphic construct. We present a synthesis of recent results which characterize this "toy" language's expressive power as well as its type reconstruction (or type inference) problem. More specifically: (1) Core-ML can express exactly the ELEMENTARY queries, where a program input is a database encoded as a λ-term and a query program is a λ-term whose application to the input normalizes to the output database. In addition, it is possible to express all the PTIME queries so that this normalization process is polynomial in the input size. (2) The polymorphism of `let` can be explained using a simple algorithmic reduction to monomorphism, and provides flexibility, without affecting expressibility. Algorithms for type reconstruction offer the additional convenience of static typing without type declarations. Given polymorphism, the price of this convenience is an increase in complexity from linear-time in the size of the program typed (without `let`) to completeness in exponential-time (with `let`). (3) Fragments of Core-ML, based on the order of functionalities used, can be fairly expressive. For example, order ≤ 5 suffices for the PTIME queries, even without `let` or equality. Also, for each fixed order, type reconstruction is polynomial in program size. Programming by using low order functionalities and type reconstruction is common in functional languages. Thus, our analysis partly explains the wide use and efficiency of such programming practices.

1 Introduction

The *simply typed λ-calculus* [11] (*typed λ-calculus* or TLC for short) with its syntax and beta-reduction operational semantics is an essential part of any functional programming language. In our presentation, we use the "Curry view" of TLC without type annotations and reconstruct *monomorphic* or *simple types*.

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

** Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Currently visiting INRIA Rocquencourt, France. Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

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

For notational convenience, we use $TLC^=$, the typed λ-calculus with atomic constants and an equality on them, and the associated delta-reduction of [4, 11]. For the necessary background on TLC see Section 2.1.

We call Core-ML the "toy" language resulting from the addition of `let`-*polymorphism* to $TLC^=$ (see Section 2.2). The primary reason for studying Core-ML is that it seems to be the essential part of the (Standard) ML language [40, 21]. One of the main features of all ML dialects, such as Standard ML, CAML, MLNJ and of other state-of-the-art functional languages such as Haskell, Miranda etc, is the combination of polymorphism (for the flexibility of code reuse) with type reconstruction (for the convenience of writing well-typed programs without declaring any types). ML's `let`-polymorphism is considered as one of the most successful definitions of polymorphism in programming languages and is widely used. Our $TLC^=$ primitives are part of every functional language. Note that, our $TLC^=$ definition is minimal, in the sense of not including the monomorphic fixpoints of ML [40, 14, 21].

The theme of our paper is: *an exploration of the Core-ML language and a demonstration that, as far as "toy" languages go, this one has nontrivial expressive power and an interesting type reconstruction problem.*

Expressive Power: The computations expressed in Core-ML (or TLC) are all ELEMENTARY functions (where this class of functions includes PTIME, NP, PSPACE, EXPTIME, 2-EXPTIME etc, see [41])[4]. We include some of the basic expressibility analysis in Section3. For more details on the analysis of expressibility we refer the reader to [44, 45, 17, 37, 23, 24, 25].

The expressive power of TLC was originally analyzed in terms of computations on simply typed Church numerals (see, e.g., [4, 17, 44]). Unfortunately, the simply typed Church numeral input-output convention imposes severe limitations on expressive power. Only a fragment of PTIME is expressible this way (called the extended polynomials) and this does not illustrate the full capabilities of TLC. That more expressive power is possible follows from the fact that provably hard decision problems can be embedded in TLC, see [39, 45, 37], and that different typings allow exponentiation [17].

One way of expressing all of PTIME, while avoiding the anomalies associated with representations over Church numerals was recently demonstrated by Leivant and Marion [35]. By augmenting the simply typed lambda calculus with a pairing operator and a "bottom tier" consisting of the free algebra of words over $\{0, 1\}$ with associated constructor, destructor, and discriminator functions, they obtained various calculi for PTIME.

In [23, 24, 25] we proposed and analysed a different approach to TLC (and Core-ML) expressibility. This is the approach we survey here. It is based on *the theory of database queries* or equivalently on *finite model theory.* Our framework is explained in detail in Sections 2.3-2.5.

Relational algebra and fixpoint formulas over finite structures have been

[4] From [41]: "The optimism in this term [ELEMENTARY] may seem a little overstated; the term was introduced in the context of undecidability."

the principal vehicles of theoretical research in database query languages; see [13, 16, 9] for some of its earlier formulations. The main motivation has been that common relational database queries are expressible in *relational calculus/algebra* [13], *Datalog*$^\neg$ and various *fixpoint logics* [3, 32, 9, 10]. Most importantly, as shown in [27, 47], every PTIME query can be expressed using fixpoints of first-order formulas on ordered structures; and, as shown in [3], it suffices to use Datalog$^\neg$ syntax under a variety of semantics to express various fixpoint logics. In addition, extensions have been proposed to this framework to manipulate complex-object databases, based on high-order formulas over finite structures, e.g., [1, 2]. The study of functional database query languages is more recent, see [6, 8, 7, 28], and is also part of the research on complex-object databases.

In [23] we showed that TLC (without additions, but with different I/O conventions) can express exactly the ELEMENTARY database queries. The basic idea was to use the framework of finite model theory instead of Church numerals. In this framework a program input is a database encoded as a λ-term and a query program is a λ-term whose application to the input normalizes to the output database. Interestingly, it is possible to express all the PTIME queries so that this normalization process is polynomial in the input size. In [24, 25] these results were refined to fragments of Core-ML of low functionality. In our presentation we discuss relational algebra, using order 3 functionalities and equality (Section3.1) and fixpoint queries, using order 4 functionalities and equality (Section3.2). The ability to encode Powerset extends these results to ELEMENTARY (Section3.3).

Type Reconstruction: There is a subtle connection between `let`-polymorphism in ML and monomorphism, which we use here as a definition. This connection has been in the folklore for some years and is now better understood [38]. We include a short, but self-contained description in Section 4.1. For a complete description of syntax and semantics we refer the reader to [40, 14, 29, 38].

The problem of Core-ML (TLC) type reconstruction is: "given a λ-term without any/some type annotations find if it is typable using the ML (TLC) typing rules". For TLC this is an efficiently solvable problem in linear-time [48], by reduction to first-order unification [43, 15], but it is also PTIME-complete. For Core-ML it is EXPTIME-complete in the size of the program typed, see [29, 30]. The proof sketch we present in Section 4.2 is based on [22].

Fixed Functionality: In [24, 25] we continued the investigation of expressibility and type reconstruction, but for low functionality orders. Fragments of Core-ML, based on the order of functionalities used, can be fairly expressive. For example, order ≤ 5 suffices for the PTIME queries, even without `let` or equality. (Note that, in this sense equality is only a notational convenience). Since Cobham's early work there have been a number of interesting *functional* characterizations of PTIME, e.g., [12, 20, 19, 5, 35]. In Section 5.1 we add to this literature. In Section 5.2, we argue that fixed order type reconstruction is in polynomial time. This has some practical significance. In Section 6 we close with some open questions.

2 Basic Definitions

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

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

The syntax of TLC λ-*terms* or *expressions* is given by the following grammar $\mathcal{E} \equiv x \mid (\mathcal{E}\mathcal{E}) \mid \lambda x.\, \mathcal{E}$, where x ranges over a set of *expression variables* (distinct from the type variables) and where expressions satisfy *typedness* as outlined below. We omit outermost parentheses and PQR stands for $(PQ)R$.

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

$$(var_M) \qquad \frac{}{\Gamma \cup \{x{:}\, \tau'\} \;\rhd\; x{:}\, \tau'}$$

$$(abs_M) \qquad \frac{\Gamma \cup \{x{:}\, \tau_0\} \;\rhd\; E{:}\, \tau_1}{\Gamma \;\rhd\; \lambda x.\, E{:}\, \tau_0 \rightarrow \tau_1}$$

$$(app_M) \qquad \frac{\Gamma \;\rhd\; M{:}\, \tau_0 \rightarrow \tau_1 \qquad \Gamma \;\rhd\; N{:}\, \tau_0}{\Gamma \;\rhd\; MN{:}\, \tau_1}$$

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

The operational semantics of TLC are defined using *reduction*. For typed λ-terms e, e', we write $e \Rightarrow_\alpha e'$ (α-reduction) when e' can be derived from e by renaming of a λ-bound variable, for example $\lambda x.\, \lambda y.\, y \Rightarrow_\alpha \lambda x.\, \lambda z.\, z$. We write $e \Rightarrow_\beta e'$ (β-reduction) when e' can be derived from e by replacing a subterm in e of the form $(\lambda x.\, E)E'$ by $[E'/x]E$ (E with E' substituted for all free occurrences of x in E). See [4] for standard definitions of substitution and reduction for both the typed and untyped λ-calculus. Let $\Rightarrow$ be the reflexive, transitive closure of $\Rightarrow_\alpha$ and $\Rightarrow_\beta$. Note that, reduction preserves types.

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

TLC$^=$: We obtain TLC$^=$ by enriching TLC with: (1) a countably infinite set $\{o_1, o_2, \ldots\}$ of atomic constants of type o (some fixed type variable), and (2) introducing an equality constant Eq of type $\mathsf{o} \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau \rightarrow \tau$ (for some fixed type variable τ different from o). The type inference rules are the same with one modification: the Γ's must treat the constants as free variables associated with the fixed types o and $\mathsf{o} \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau \rightarrow \tau$, respectively. The reduction rules of TLC$^=$ are obtained by enriching the operational semantics of TLC as follows.

For every pair of constants o_i, o_j: $\mathtt{o}$, we add to $\Rightarrow$ the reduction rule (known as delta reduction):

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

A λ-term from which no reduction is possible is in *normal form*. TLC and TLC$^=$ enjoy the following properties, see [11, 4]:

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

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

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

Type Reconstruction: One can show that given E it is decidable whether E is a typed λ-term and the principal type of this term can be reconstructed. Also, given $\Gamma \rhd E{:}\,\tau'$ it is decidable if this statement is derivable by the above rules. (Both these algorithms use first-order unification, e.g., see [29] and reconstruct types. They work with or without type annotations and with or without constants in the Γ's).

TLC$^=$ type reconstruction is *linear-time* in the size of the program analysed.

2.2 Core-ML = TLC$^=$ + `let`-Polymorphism

Core-ML: The syntax is TLC$^=$ augmented with one new expression construct: `let` $x = \mathcal{E}$ `in` $\mathcal{E}$ and with an ML-typedness requirement, instead of a typedness requirement.

ML-typedness: This involves the same monomorphic types and inference rules for TLC and the constants used for TLC$^=$, with one additional rule that captures some *polymorphism* (see [29]):

$$(\mathtt{let}_M) \qquad \frac{\Gamma \rhd E{:}\,\tau_0 \qquad \Gamma \rhd [E/x]B{:}\,\tau}{\Gamma \rhd \mathtt{let}\ x = E\ \mathtt{in}\ B{:}\,\tau}$$

We call a λ-term E *ML-typed* if $\Gamma \rhd E{:}\,\tau'$ is derivable by the inference rules (var_M), (abs_M), (app_M), $(\mathtt{let}_M)$ for some Γ and τ'. One consequence is that TLC$^=$ is a subset of Core-ML.

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

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

Note that, Core-ML has all the properties of TLC$^=$, i.e., Church-Rosser, Strong Normalization, Principal Type and Type Reconstruction. There is only one difference: Type reconstruction is no longer in linear-time but EXPTIME-complete [29, 30].

2.3 Functionality Order

For a finer analysis of expressibility we partition types according to functionality order. The *order* of a type measures the higher-order functionality of a λ-term of that type: $o(t) = 0$ for a type variable t and $o(\tau' \to \tau'') = \max(1 + o(\tau'), o(\tau''))$. We also refer to the order of a typed λ-term as the order of its type. Note that, the order of the fixed type variables o and τ is 0.

The above definitions and properties hold for fragments of TLC, TLC$^=$, and Core-ML, where order of terms is some fixed k. In such fragments we use the above inference rules (var_M), (abs_M), (app_M), $(\mathtt{let}_M)$ but with all types restricted to order k.

2.4 The Basic Programming Technique: List Iteration

List iteration is a powerful programming technique, which can be used in the context of TLC and TLC$^=$ to encode any elementary recursion [45, 37]. However, some care is needed if one is to maintain typedness [23]. We briefly review how list iteration works. Let $\{x_1, x_2, \ldots, x_k\}$ be a set of λ-terms, each of type α; then

$$L \equiv \lambda c\colon \alpha \to \tau' \to \tau'.\, \lambda n\colon \tau'.\, c\, x_1\, (c\, x_2 \ldots (c\, x_k\, n) \ldots)$$

is a λ-term of type $(\alpha \to \tau' \to \tau') \to \tau' \to \tau'$, for *any* type τ'—in other words, L is a typable term no matter what type τ' we choose (though one fixed term must be chosen when we compute). We abbreviate this list construction as $[x_1, x_2, \ldots, x_k]$; the variables c and n abstract over the constructors *cons* and *nil*.

For example, a standard coding of Boolean logic uses $\mathit{true} \equiv \lambda x\colon \tau.\, \lambda y\colon \tau.\, x$ and $\mathit{false} \equiv \lambda x\colon \tau.\, \lambda y\colon \tau.\, y$, both of type $\mathsf{Bool} \equiv \tau \to \tau \to \tau$. Define the exclusive or as $\mathit{Xor} \equiv \lambda p\colon \mathsf{Bool}.\, \lambda q\colon \mathsf{Bool}.\, \lambda x\colon \tau.\, \lambda y\colon \tau.\, p(qyx)(qxy)$, and the parity of a list of Boolean values as

$$\begin{aligned}&\mathit{Parity} \equiv \lambda L\colon (\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool}.\\ &\quad L\, \mathit{Xor}\, \mathit{false}.\end{aligned}$$

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.

2.5 The I/O Conventions: Databases as λ-terms

In order to study the expressive power of Core-ML (or equivalently TLC$^=$) we must fix the I/O conventions for computations. The ones we use here are similar with those in [23], but more economical in order of functionality. Inputs and outputs are encodings of a set of relations (or a first-order structure or a finite model or a database). Relations are represented as follows. Let $O = \{o_1, o_2, \ldots\}$ be the set of constants of the TLC$^=$ calculus. For convenience, we assume that this set of constants also serves as the universe over which relations are defined.

Let $r = \{(o_{1,1}, o_{1,2}, \ldots, o_{1,k}), (o_{2,1}, o_{2,2}, \ldots, o_{2,k}), \ldots, (o_{m,1}, o_{m,2}, \ldots, o_{m,k})\} \subseteq O^k$ be a k-ary relation over O. An *encoding* $\overline{r}$ of r is the λ-term

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

If r contains at least two tuples, then the principal type of $\overline{r}$ is $(\mathsf{o} \to \cdots \to \mathsf{o} \to t \to t) \to t \to t$, where t is a free type variable. (If r is empty or contains only one tuple, this type is only an instance of the principal type of $\overline{r}$.) The order of this type is 2, independent of the arity of r. We abbreviate this type as o^t_k. Instances of this type, obtained by substituting some type expression θ for t, are abbreviated as o^θ_k, 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^t_k characterizes these terms:

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

We can now define various query languages as fragments of Core-ML.

Definition 2. A query program of arity $(k_1, \ldots, k_l, k)$ in $TLI^=_i$ (the language of *typed list iteration of order $i+3$ with equality*) is a typed $\mathrm{TLC}^=$ 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 $\overline{r_1} \ldots \overline{r_l}$ one can type $(\lambda R_1 \ldots \lambda R_l.\, M)\overline{r_1} \ldots \overline{r_l}$ as o^τ_k.

Remark: In the setting of these query languages input and output terms are monomorphically typed, i.e., they have simple types which are the same for all inputs/outputs. However, unlike [17, 44], we allow that the monomorphic types of inputs and outputs differ. Outputs are always typed as o^τ_k but inputs can be typed as o^*_k. This convention is necessary for expressing all of PTIME.

The above query language definitions are semantic because they involve quantification over all inputs. By Lemma 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.

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

An analogous set of query languages: $MLI^=_i$ (the languages of *typed list iteration of order $i+3$ with equality*) can be defined using Core-ML instead $\mathrm{TLC}^=$ and ML-typedness instead of typedness (see [24]).

3 Core-ML Expressibility

Recall that Core-ML and TLC$^=$ have the same expressive power. In this section we will examine the expressibility of various TLI$_i^=$ fragments of these languages. We will use expressibility results and terminology from the theory of database queries, e.g., [9].

We first argue that TLI$_0^=$ can express relational algebra queries and then that TLI$_1^=$ can express fixpoints (on ordered structures) or PTIME queries. It turns out, although we do not outline the argument here, that these are exact characterizations of these fragments, see [24, 25]. By increasing the order of functionality one can express all ELEMENTARY queries and no other queries are expressible.

3.1 Embedding Relational Algebra in TLI$_0^=$

In [23], we showed how to express relational algebra (i.e., the operations of Select, Project, Difference, Union, Intersection, Times) using list iteration. Due to a different input/output format, our encodings involved λ-terms of order 5. With straightforward modifications, these terms work under the present input/output conventions and the order drops down to 3.

Here, we give the Cartesian product (Times) and Intersection operators as examples and refer the reader to [23] for the other operators.

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

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

where

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

Using list iteration one can program every relational algebra operator in a similar well-typed fashion.

These relational operator λ-terms when applied to one (resp., two) relations coded in the format described in the previous section reduce to a normal form which is the encoding of the desired output relation. Moreover, when the terms are reduced according to a fairly simple reduction strategy, 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. From [23], we have the following theorem.

Theorem 4. *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 Core-ML ($TLI_0^=$) 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 $\phi(r_1, \ldots, r_l)$, where $\overline{r}$ denotes the encoding of relation r. The normal form of $(\psi\, \overline{r_1} \ldots \overline{r_l})$ can be computed in time and reductions polynomial in $r_1, \ldots, r_l$.*

$\mathrm{TLI}_0^=$ is not powerful enough to go beyond the relational algebra and compute fixpoints of relational queries. This is because the language only allows the iteration of mappings from order-zero objects to order-zero objects. It is necessary to go to $\mathrm{TLI}_1^=$ so as to iterate mappings from relations to relations.

3.2 Embedding PTIME/Fixpoint Queries in $\mathrm{TLI}_1^=$

That $\mathrm{TLI}_1^=$ is sufficient to express the Fixpoint queries (and by [27, 47] the PTIME queries) follows from the encodings given in [23], plus the fact that over a known domain, relations can be represented by order-one objects, namely *characteristic functions*.

The characteristic function f_r of a k-ary relation r is a $\mathrm{TLC}^=$ term of type $\mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{Bool}$, such that for any k constants $o_{i_1}, \ldots, o_{i_k}$,

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

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

$$\begin{aligned}
&\Upsilon \equiv \lambda R_1 \ldots \lambda R_l.\\
&\quad \mathit{FuncToList}\\
&\quad\quad (\mathit{Crank}\\
&\quad\quad\quad (\lambda \mathbf{x}.\, \lambda f.\, \mathit{ListToFunc}\, (Q\, (\mathit{FuncToList}\, f)))\\
&\quad\quad\quad (\mathit{ListToFunc}\; \mathit{Nil})),
\end{aligned}$$

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

As explained in [23], additional care is necessary to make Υ typable using monomorphic types. This is because the inputs $R_1, \ldots, R_l$ are used to iterate both over order-one objects (in *Crank*) and over order-zero objects (in Q). With monomorphic types, this is normally impossible. However, [23] shows how to get around this problem by introducing a "type-laundering" operator that essentially turns iterations over order-zero objects into iterations over order-one objects. By using this operator inside Q, the term Υ becomes typable in the monomorphic type system.

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

$$
\begin{array}{l}
\texttt{let } R_1 = \overline{r_1} \texttt{ in} \ldots \texttt{let } R_l = \overline{r_l} \texttt{ in} \\
\quad \mathit{FuncToList} \\
\qquad (\mathit{Crank} \\
\qquad\quad (\lambda \mathbf{x}.\, \lambda f.\, \mathit{ListToFunc}\,(Q\,(\mathit{FuncToList}\, f))) \\
\qquad\quad (\mathit{ListToFunc}\, \mathit{Nil}))\,,
\end{array}
$$

where $\overline{r_1}, \ldots, \overline{r_l}$ are the encodings of the input relations, the variables $R_1, \ldots, R_l$ are declared to be polymorphic, so it does not matter that their occurrences in *Crank* and Q require different types.

Theorem 5. *Let $\phi(R_1, \ldots, R_l)$ be a fixpoint query over relational schema $\mathcal{S} = (R_1, \ldots, R_l)$. Then there exists a term ψ of Core-ML ($TLI_1^=$) 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. The normal form of $(\psi\, \overline{r_1} \ldots \overline{r_l})$ can be computed in time and reductions polynomial in the size of $r_1, \ldots, r_l$.*

3.3 Elementary Functions = Core-ML

In [17], a bound is given that any λ-term of size s and order d has a normal form of size at most $e(s, d)$ where $e(a, 0) = a$, $e(a, b + 1) = 2^{e(a,b)}$. The ELEMENTARY functions are those contained in union over d of DTIME$[e(n, d)]$. It follows that Core-ML can only express such functions. This is no real limitation, since all practical functions are ELEMENTARY and, as shown in [23], all ELEMENTARY functions are expressible. This was shown using an alternative characterization of the Abiteboul-Beeri (AB) complex object algebra, [1].

Theorem 6. *Let $\phi(R_1, \ldots, R_l)$ be an AB complex object algebra expression over the relational schema $\mathcal{S} = (R_1, \ldots, R_l)$. Then there exists a term ψ of Core-ML ($TLC^=$) 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. If the Powerset operator is not used in ϕ, the normal form of $(\psi\, \overline{r_1} \ldots \overline{r_l})$ can be computed in time and reductions polynomial in the size of $r_1, \ldots, r_l$.*

4 Core-ML Type Reconstruction

4.1 Polymorphism vs Monomorphism

The syntax of *polymorphic types* is given by the grammar

$$\mathcal{T}_0 ::= t \mid \mathcal{T}_0 \to \mathcal{T}_0$$
$$\mathcal{T} ::= \mathcal{T}_0 \mid \forall t.\, \mathcal{T}$$

where t ranges over a set of *type variables.* A type is just a monotype (as defined in Section 2), with possibly some quantification over type variables at the outermost level. We refer to $\tau \in \mathcal{T}_0$ as *monotypes*, and $\sigma \in \mathcal{T}$ as *polytypes* (sometimes also called *type schemes*).

We interpret polytypes as sets of monotypes, using the interpretation: $\langle \alpha \rangle = \{\alpha\}$ where α is a monotype, and $\langle \forall t.\, \alpha \rangle = \bigcup_{\tau \in \mathcal{T}_0} \langle [\tau/t]\alpha \rangle$, where $[\tau/t]\alpha$ denotes the substitution of τ for free occurrences of t in α. The interpretation of polytypes as sets of monotypes allows the definition of a partial order $\sqsubseteq$ on $\mathcal{T}$: we write $\sigma_1 \sqsubseteq \sigma_2$ iff $\langle \sigma_2 \rangle \subseteq \langle \sigma_1 \rangle$. It is easy to see, for instance, that $\forall t.\, \alpha \sqsubseteq [\tau/t]\alpha$ for any polytype α and monotype τ. Note the minimal element of $\mathcal{T}$ is $\forall t.\, t$, since $\langle \forall t.\, t \rangle = \mathcal{T}_0$.

Expressions are associated with types using a fixed set of *inference rules.* We describe the standard standard inference system given by Damas and Milner [14] which we call the *polytype system*, as distinguished from the earlier-described *monotype system.* We want to identify exactly in what sense the implied restrictiveness of the monotype system is a constraint on expressiveness.

Core ML inference rules for the polytype system

(var_P)
$$\frac{}{\Gamma \cup \{x{:}\,\sigma\} \ \triangleright\ x{:}\,\sigma}$$

(gen_P)
$$\frac{\Gamma \ \triangleright\ E{:}\,\sigma(t) \qquad [t \notin \mathsf{FV}(\Gamma)]}{\Gamma \ \triangleright\ E{:}\,\forall t.\, \sigma(t)}$$

$(inst_P)$
$$\frac{\Gamma \ \triangleright\ E{:}\,\forall t.\, \sigma(t)}{\Gamma \ \triangleright\ E{:}\,[\tau/t]\sigma}$$

(abs_P)
$$\frac{\Gamma \cup \{x{:}\,\tau_0\} \ \triangleright\ E{:}\,\tau_1}{\Gamma \ \triangleright\ \lambda x.\, E{:}\,\tau_0 \to \tau_1}$$

(app_P)
$$\frac{\Gamma \ \triangleright\ M{:}\,\tau_0 \to \tau_1 \qquad \Gamma \ \triangleright\ N{:}\,\tau_0}{\Gamma \ \triangleright\ MN{:}\,\tau_1}$$

$(\mathtt{let}_P)$
$$\frac{\Gamma \ \triangleright\ E{:}\,\sigma \qquad \Gamma \cup \{x{:}\,\sigma\} \ \triangleright\ B{:}\,\tau}{\Gamma \ \triangleright\ \mathtt{let}\ x = E\ \mathtt{in}\ B{:}\,\tau}$$

We refer to the *monotype system*, as defined in Section 2, as only having rules (var_M), (abs_M), (app_M), or $(\mathtt{let}_M)$, each identical to their polytype counterpart above, except that the environments Γ contain monotypes only. Write $\Pi \vdash_P \Gamma \triangleright$

$E{:}\,\tau$ to indicate that Π is a derivation in the polytype system, and $\Pi \vdash_M \Gamma \rhd E{:}\,\tau$ similarly for the monotype system. When Π is omitted, it is merely the existence of a derivation that is asserted.

The polytype system has the well known *principal type property:* for each ML expression E, there is a context Γ and type π where $\vdash_P \Gamma \rhd E{:}\,\pi$, and for any type τ, $\vdash_P \Gamma \rhd E{:}\,\tau$ implies $\pi \sqsubseteq \tau$.

We now clarify in what sense the polytype and monotype systems are equivalent, by relating the polymorphic and monomorphic interpretation of `let`:

Theorem 7. *(Equivalence Theorem) Let* $\Gamma = \{w_1{:}\,\alpha_1, w_2{:}\,\alpha_2, \ldots, w_m{:}\,\alpha_m\}$ *be any context of monotype bindings, and* $\Gamma' = \{y_1{:}\,\beta_1, y_2{:}\,\beta_2, \ldots, y_n{:}\,\beta_n\}$ *be any context of polytype bindings. Let* $\mathcal{F} = \{F_1, F_2 \ldots, F_n\}$ *be a set of ML expressions where* β_j *is the principal type of* F_j*; specifically, we insist that for* $1 \leq j \leq n$*,*

$$\vdash_P \Gamma \cup \{y_1{:}\,\beta_1, y_2{:}\,\beta_2, \ldots, y_{j-1}{:}\,\beta_{j-1}\} \rhd F_j{:}\,\beta_j$$
$$\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j{:}\,\overline{\beta_j}$$

where $\overline{\beta_j}$ *is* β_j *with all quantifiers removed. Let* E *be any term and* τ *be any monotype; then*

$$\vdash_P \Gamma \cup \Gamma' \rhd E{:}\,\tau \text{ if and only if } \vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]E{:}\,\tau$$

Proof. Note that in the case of a closed term E with empty contexts, we have $\vdash_P \emptyset \rhd E{:}\,\tau$ iff $\vdash_M \emptyset \rhd E{:}\,\tau$, as in [29]. However, inspired by the example of Tait's strong normalization theorem for the first order typed λ-calculus [46], we facilitate the proof by strengthening the induction hypothesis of what is to be a syntax directed induction on E.

We first introduce a standard structural lemma allowing us to "normalize" derivations in the polytype system for use in a syntax-directed proof. See, for example, [38] for the proof.

Lemma 8. *Let* $\Pi \vdash_P \Gamma \rhd E{:}\,\forall.\,\tau$ *where* Γ *is a context,* τ *is a monotype, and* $\forall$ *denotes a (possibly empty) list of quantified variables. Then if* E *is not a variable, there exists a proof* $\Pi' \vdash_P \Gamma \rhd E{:}\,\tau$ *where the last rule used in* Π' *is either* (var_P)*,* (abs_P)*,* (app_P)*, or* ($\mathtt{let}_P$)*.*

Given the lemma and the stated assumptions on Γ, Γ', and $\mathcal{F}$, we prove the Theorem via structural induction on E, proceeding by case analysis. We assume by renaming that no variable is bound or quantified more than once.

Case $E \equiv w_i$ Then necessarily $\tau \equiv \alpha_i$ and $\vdash_P \Gamma \cup \Gamma' \rhd w_i{:}\,\alpha_i$; since we know that $[F_1/y_1][F_2/y_2]\cdots[F_n/y_n]w_i \equiv w_i$, the result is immediate.

Case $E \equiv y_j$ In the forward direction, assume $\vdash_P \Gamma \cup \Gamma' \rhd y_j{:}\,\tau$. Since $\vdash_P \Gamma \cup \Gamma' \rhd y_j{:}\,\beta_j$ is a principal typing, we know $\beta_j \sqsubseteq \tau$; since $\tau \in \mathcal{T}_0$, we know there exists a substitution Σ such that $\Sigma\overline{\beta_j} = \tau$. But since

$$[F_1/y_1][F_2/y_2]\cdots[F_n/y_n]y_j \equiv [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j,$$

and

$$\Pi \vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j\colon \overline{\beta_j}$$

we know

$$\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j\colon \tau$$

by applying Σ to all types appearing in the proof Π. In the reverse direction, suppose $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j\colon \tau$; then by principality we know $\Sigma\overline{\beta_j} = \tau$. Given $\vdash_P \Gamma \cup \Gamma' \rhd y_j\colon \beta_j$, we derive $\vdash_P \Gamma \cup \Gamma' \rhd y_j\colon \tau$ by instantiating the $\forall$-bound variables of β_j according to Σ.

Case $E \equiv GH$ Straightforward.

Case $E \equiv \lambda x.\, G$ To prove "only if," if $\vdash_P \Gamma \cup \Gamma' \rhd \lambda x.\, G\colon \tau' \to \tau''$ where $\tau \equiv \tau' \to \tau''$, then by Lemma 8 we know $\vdash_P \Gamma \cup \{x\colon \tau'\} \cup \Gamma' \rhd G\colon \tau''$. By induction on G (with a larger monotype context, since the binding for x is added), we have $\vdash_M \Gamma \cup \{x\colon \tau'\} \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G\colon \tau''$, and by (abs_M), $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]\lambda x.\, G\colon \tau' \to \tau''$. All implications are reversible.

Case $E \equiv \texttt{let}\ y = G\ \texttt{in}\ H$ If $\vdash_P \Gamma \cup \Gamma' \rhd \texttt{let}\ y = G\ \texttt{in}\ H\colon \tau$, then by Lemma 8 and $(\texttt{let}_P)$, $\vdash_P \Gamma \cup \Gamma' \rhd G\colon \sigma$ for (principal) polytype σ, and $\vdash_P \Gamma \cup \Gamma' \cup \{y\colon \sigma\} \rhd H\colon \tau$. By $(inst_P)$, $\vdash_P \Gamma \cup \Gamma' \rhd G\colon \overline{\sigma}$, so by induction on G we have a proof

$$\Pi \vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G\colon \overline{\sigma}.$$

Hence by induction on H with polytype context $\{y_1\colon \beta_1, y_2\colon \beta_2, \ldots, y_n\colon \beta_n, y\colon \sigma\}$ and associated code $\{F_1, F_2, \ldots, F_n, G\}$, we have

$$\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n][G/y]H\colon \tau,$$

which we rewrite as

$$\vdash_M \Gamma \rhd [[F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G/y][F_1/y_1][F_2/y_2]\cdots[F_n/y_n]H\colon \tau.$$

Using proof Π above and $(\texttt{let}_M)$, we then have a proof of

$$\vdash_M \Gamma \rhd \texttt{let}\ y = [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G\ \texttt{in}\ [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]H\colon \tau,$$

which is syntactically identical to

$$\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]\texttt{let}\ y = G\ \texttt{in}\ H\colon \tau.$$

Once again, the argument is reversible.

This concludes the proof of the equivalence theorem.

Corollary 9. *Let E be a closed term and τ be a monotype. Then $\vdash_P \emptyset \rhd E\colon \tau$ if and only if $\vdash_M \emptyset \rhd E\colon \tau$.*

4.2 Core-ML Type Reconstruction is EXPTIME-complete

While the proof of this claim is somewhat detailed, it can easily be summarized in a short paragraph. Dwork, Kanellakis, and Mitchell proved that first-order unification was complete for PTIME, by showing how unification on first-order terms could simulate circuits [15]; the proof followed by reduction from the Circuit Value Problem [34]. Type inference for ML programs without `let` (equivalently, simply-typed lambda calculus) is easily reduced to first-order unification, where a program is transformed to a unification problem of size linear in the size of the original program [48] and vice versa. As a consequence, type inference for the simply-typed case is complete for PTIME. By adding the polymorphic `let` and iterating its use, it is possible to write a program that generates a unification problem of *exponential* size. Bootstrapping the ideas of [15], one may then simulate an exponential-size circuit and use standard complexity theory techniques [41].

For realism, we use genuine Standard ML scripts (with compiler-generated type information) to outline the proof, which can in fact be fully automated. The Standard ML syntax `fn x=> e` is the equivalent of the λ-calculus $\lambda x.\, e$, and the `let` statement is delimited, e.g., `let x=e in b end`. Each line beginning with `val` (value) or `fun` (function) is meant to be a nested `let`, where `let val x= e` means `let x=e in ...`; the syntax `let fun F x y z= e` is syntactic sugar for `let val F= fn x=> fn y=> fn z=> e`. To avoid conflict with ML reserved words, examples using ML capitalize the names of declared functions.

Unification is PTIME-complete We first show how the theorem of [15] could have been proved by writing an ML program using the half-century old, classic λ-calculus encodings of the logical operations. Here are Church's standard definitions of the Boolean constants and functions, coded in ML.

The Standard ML compiler responds with typings of the expressions entered by the user. The variables `'a`, `'b`, ... refer to *outermost quantified* type variables, so (for example) the type `'a -> 'b -> 'a` of `True` is $\forall a.\, \forall b.\, a \rightarrow b \rightarrow a$.

```
- fun True x y= x;
val True = fn : 'a -> 'b -> 'a
- fun False x y= y;
val False = fn : 'a -> 'b -> 'b
- fun And p q= p q False;
val And = fn : ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a -> 'd
- fun Or p q= p True q;
val Or = fn : (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c -> 'd
- fun Not p= p False True;
val Not = fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e) -> 'e
```

When these Boolean functions are used, notice that they output functions as values; moreover, the principal types of these functions identify them uniquely as `True` or `False`:

```
- Or False True;
val True = fn : 'a -> 'b -> 'a
- And True False;
val False = fn : 'a -> 'b -> 'b
- Not True;
val False = fn : 'a -> 'b -> 'b
```

While the compiler does not *necessarily* reduce the above expressions to normal form, hence computing an "answer," its type inference mechanism implicitly carries out that reduction to normal form, expressed in the language of first-order unification. We now introduce pairing and fanout:

```
- fun Pair x y= fn z=> z x y;
val Pair = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun Fanout p= p (Pair True True) (Pair False False);
val Fanout = fn : (((('a -> 'b -> 'a) -> ('c -> 'd -> 'c) -> 'e) -> 'e)
-> ((('f -> 'g -> 'g) -> ('h -> 'i -> 'i) -> 'j) -> 'j) -> 'k) -> 'k
```

The importance of **Fanout**, as in [29], is that it produces two copies of a logic value which do not share type variables.

```
- Fanout True;
val it = fn : (('a -> 'b -> 'a) -> ('c -> 'd -> 'c) -> 'e) -> 'e
- Fanout False;
val it = fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'd) -> 'e) -> 'e
```

We code circuits so that every Boolean value is used *exactly once*.

A Boolean circuit can be coded as a λ-term by labelling its (wire) edges and traversing them bottom-up, inserting logic gates and fanout gates appropriately. We consider the circuit example from ([15], p. 43), and realized by the straight-line pseudocode. (Here, lambdas mark inputs, and semicolons mark sequential computations; the line $\langle e_9, e_{10}\rangle = \mathit{fanout}\,(\mathit{and}\, e_7\, e_8)$ should be read as "compute $\mathit{and}\, e_7\, e_8$ and fan out two copies (wires) with the result, labelled e_9 and e_{10}." The output of the circuit is $\mathit{or}\, e_{11}\, e_{12}$.)

$$
\begin{aligned}
&\lambda \mathrm{e}_1 .\, \lambda \mathrm{e}_2 .\, \lambda \mathrm{e}_3 .\, \lambda \mathrm{e}_4 .\, \lambda \mathrm{e}_5 .\, \lambda \mathrm{e}_6 . \\
&\qquad\qquad\qquad\qquad e_7 = \mathit{and}\; e_2\, e_3; \\
&\qquad\qquad\qquad\qquad e_8 = \mathit{and}\; e_4\, e_5; \\
&\qquad\qquad\qquad \langle e_9, e_{10}\rangle = \mathit{fanout}\;(\mathit{and}\; e_7\, e_8); \\
&\qquad\qquad\qquad\qquad e_{11} = \mathit{or}\; e_1\, e_9; \\
&\qquad\qquad\qquad\qquad e_{12} = \mathit{or}\; e_{10}\, e_6; \\
&\qquad\qquad\qquad \mathit{or}\; e_{11}\, e_{12}
\end{aligned}
$$

Removing the syntactic sugar, this code "compiles" to the slightly less comprehensible

```
- fun Circuit e1 e2 e3 e4 e5 e6=
(fn e7=>
  (fn e8=>
```

```
    (Fanout (And e7 e8))
      (fn e9=> fn e10=>
        (fn e11=>
          (fn e12=> Or e11 e12)
          (Or e10 e6))
        (Or e1 e9)))
  (And e4 e5))
(And e2 e3);

val Circuit = fn : (('a -> 'b -> 'a) -> 'c -> ('d -> 'e -> 'd) -> 'f
-> 'g) -> ('h -> ('i -> 'j -> 'j) -> 'k -> ('l -> 'm -> 'm) -> ((('n
-> 'o -> 'n) -> ('p -> 'q -> 'p) -> 'r) -> 'r) -> ((('s -> 't -> 't)
-> ('u -> 'v -> 'v) -> 'w) -> 'w) -> ('c -> (('x -> 'y -> 'x) -> 'z ->
'f) -> 'g) -> 'ba) -> 'h -> ('bb -> ('bc -> 'bd -> 'bd) -> 'k) -> 'bb
-> 'z -> 'ba
```

The type of `Circuit` is the equivalent of the construction in [15] of the circuit as a first-order term. We can compute circuit values by instantiating the inputs appropriately, for instance:

```
- Circuit False True True True True False;
val it = fn : 'a -> 'b -> 'a
```

Observe that this evaluation produces both the correct type, and the correct value: there is of course only one closed λ-term in normal form with the given type, namely $true \equiv \lambda x.\, \lambda y.\, x$. The computation of the value is performed by the interpreter, while that of the type is performed by the compiler, yet both are essentially the same. In essence, we have forced the compiler—more specifically, the type inference mechanism—into doing computation typically carried out by the interpreter. It should be clear that any Boolean circuit can be transformed into such a λ-term.

In mundane programming language terminology, the complexity theoretic *reduction* is merely a *compiler*. The size of the λ-term is clearly linear in the size of the circuit, and the transformation described can be effected in polynomial time. Observe that polynomial space is required by this translation scheme, since output wire names (for example, `e7`) are output by the transducer while their *values* (for example, `And e2 e3`) are pushed on a stack for subsequent output. The size of the stack can clearly be linear in the size of the ML program output by the transducer.

We can in fact carry out this sort of reduction in logarithmic space, curiously, by coding computation in a continuation-passing style. (A hint towards carrying out such a reduction is given by the use of the `Fanout` gate in the above example.) Code a continuation-passing version of binary logical functions, to be used in implementing real straight-line code:

```
- fun CP-style fnc p q k= k (fnc p q);
val CP-style = fn : ('a -> 'b -> 'c) -> 'a -> 'b -> ('c -> 'd) -> 'd
- fun Circuit e1 e2 e3 e4 e5 e6=
    (CP-style And) e2 e3 (fn e7=>
```

```
    (CP-style And) e4 e5 (fn e8=>
    (CP-style And) e7 e8 (fn f=>
    Fanout f (fn e9=> fn e10=>
    (CP-style Or) e1 e9 (fn e11=>
    (CP-style Or) e10 e6 (fn e12=>
    Or e11 e12))))));
val Circuit = fn : < ... type omitted ... >
```

A trivial analysis shows this translation scheme to be a logarithmic space reduction, since the transducer need only *count* right parentheses to be output at the end of the expression, instead of storing expressions on a stack. In this analysis, we have also assumed that the wire names have length logarithmic in the size of the circuit.

`Let`-polymorphism, function composition, exponential bounds Now we make an obvious observation: the transition function of a Turing machine is just a Boolean circuit. As such, we should be able to define a `let`-free ML function (i.e., simply-typed λ-term) `delta` realizing the function.[5] The salient property of `delta` is: if `ID` and `ID'` are ML terms coding successive Turing machine IDs, then `delta ID` reduces to `ID'`, and `delta ID` and `ID'` have the same ML type. In other words, merely *typing* `delta ID` requires computing the type encoding the next ID of M. Because ML codings of different IDs receive different types (just as the different Boolean values `True` and `False` have different types), this property assures us that unification and type inference, just as in the simple Boolean calculations above, unambiguously simulate computation via reduction of expressions.

Given such a realization of the transition function of a Turing machine by an ML term `delta`, we use the polymorphic `let` to succinctly (i.e., in terms of program length) iterate applications of `delta`. We first define *polymorphic* function composition:

```
- fun compose f g= fn x=> f (g x);
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Since the whole idea of this proof technique is that unification can be used to simulate computation, `compose` can be used to chain together computations (at the level of types): the output of `g` is forced to unify with the input of `f`. `Compose` is used to iterate `delta`:

```
- val δ_k =
     let val δ_0 = delta in
             let val δ_1= compose δ_0 δ_0 in
                  let val δ_2= compose δ_1 δ_1 in
                       ...
                           let val δ_{k-1}= compose δ_{k-2} δ_{k-2} in
                                compose δ_{k-1} δ_{k-1} end end ... end
```

[5] We omit the technical details of coding; the interested reader should consult [29, 22].

The function δ_k applies `delta` 2^k times to its argument, and moreover has a definition of length $O(k)$. Notice that `compose` is used to compose a function ϕ with itself, even though the types of the domain and range of ϕ are different. *Polymorphism* is the key: by appealing to the monomorphic `let` rule ($\texttt{let}_M$) and Theorem 7, each *copy* of ϕ has distinct type variables, and unification forces the domain of one ϕ to unify with the range of the other ϕ. (By contrast, a monomorphic function of type, say, `int -> bool` cannot be composed with itself.) This idea is clarified by considering the composition of `Not` with itself, instead of the more complex `delta`:

```
- val Not2= compose Not Not;
val Not2 = fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) ->
('e -> 'f -> 'f) -> ('g -> 'h -> 'g) -> 'i) -> 'i
- Not2 True;
val it = fn : 'a -> 'b -> 'a
```

Theorem 10. *Recognizing the typable Core ML expressions is* DTIME$[2^{n^t}]$*-hard for any integer* $t \geq 1$ *under logspace reduction.*

Proof. Using coding techniques described above (see [22, 29] for more details), we code in ML a transition function `delta` of an arbitrary Turing machine M, an arbitrary initial configuration `initial-ID` for M, and a function `accept?` where `accept? ID` has the type of `True` if `ID` codes an accepting ID of M, and has the type of `False` otherwise. Now we define a "testing" function `Test`:

```
- fun Test p z w= p z w w;
val Test = fn : ('a -> 'b -> 'b -> 'c) -> 'a -> 'b -> 'c
- Test True;
val it = fn : ('a -> 'b) -> 'a -> 'b
- Test False;
Error: operator and operand don't agree (circularity)
  operator domain: 'Z -> 'Y -> 'Y -> 'X
  operand:         'Z -> 'Y -> 'Y
  in expression:
    Test False
```

It is straightforward to show that `Test (accept? (`δ_k` initial-ID))` is typable iff M accepts the initial configuration in 2^k steps, since in the case of acceptance, `accept? (`δ_k` initial-ID)` has the same type as `True`, and otherwise has the same type as `False`. If n is the length of the initial tape configuration, letting $k = n^t$ for any fixed integer $t \geq 1$ proves the theorem.

5 PTIME Expressibility/Reconstruction

Fragments of Core-ML, based on the order of functionalities used, can be fairly expressive. In this section we argue that fixed order (≤ 5) suffices for the PTIME queries, even without `let` or equality. Also, for each fixed order, type reconstruction is polynomial in program size.

Since programming by using low order functionalities and type reconstruction is common in functional languages, the arguments presented here partly explain the wide use and efficiency of such programming practices.

5.1 Eliminating `let` and Equality: PTIME in TLI_2

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

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

and equality is encoded as

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

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

Relations are encoded as iterators in the usual way, except that explicit constants are replaced by the corresponding projection functions. Note that the arity of the projection functions changes with the size of the database universe, so the encoding of a relation r depends not only on r itself, but also on the database that r appears in. The same goes for the equality predicate: different databases may need different encodings of Eq. Hence, *in this setting Eq has to be part of the input.* We call the resulting language (with the modified input-output conventions) TLI_2.

The encoding of fixpoint queries described in Section 3.2 works unchanged in this new setting, except that the symbol Eq has to be λ-bound at the outermost level. The order of the query terms increases by 1, because the characteristic function of a relation now becomes an order 2 object (mapping k projection functions to a Boolean). It follows that

Theorem 11. *Every PTIME query is expressible in TLI_2.*

5.2 Fixed Order Functionality and PTIME Type Reconstruction

As shown in Section3.2, ML polymorphism can provide some flexibility in programming fixpoints. This was just an example of the fair amount of flexibility provided by the `let` construct (which is a very common programming feature).

The `let` construct is used in the various MLI's of [24] to receive the inputs, but also can be used in the body of the program. The occurrences of `let` in the program body can be eliminated by reduction at the expense of program body length [29].

A problem with use of `let` in the program body is that type inference may become inefficient. We show, however, that the fixed order restriction can be used to eliminate this inefficiency.

Theorem 12. *For each fixed k, type inference in order k Core-ML$^=$ is polynomial in the program size.*

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

6 Conclusions and Open Problems

We have overviewed the state-of-the-art of expressibility analysis for Core-ML (TLC and TLC$^=$) as well as of the problem of type reconstruction in the ML type discipline.

Much is now known about expressibility, but some interesting problems remain open. (a) Determine the exact expressive power of the various fragments of Core-ML. Analogous results exist for higher orders in other languages see [31, 26, 33]. Some progress on this is reported in [25]. involving functional characterizations of other complexity classes, in particular PSPACE and EXPTIME. (2) Study optimal reduction strategies [36] in the TLC. (3) Study languages that combine list iterators and set iterators ala [6, 8, 7, 28].

One consequence of the limitations of computing on Church numerals was a focus on the expressive power of typed calculi with more primitives and more complex type disciplines. Examples are the Girard-Reynolds second-order typed λ-calculus or System F [18, 42] (adding type polymorphism via type quantification) or Milner's ML [40] (adding monomorphic fixpoints and `let`-polymorphism). However, the expressibility analysis here indicates that the case for more powerful programming languages should be made on the basis of flexibility and not expressibility (since all PTIME queries can be representable in TLC). One should perhaps re-examine what algorithms cannot be described in the typed λ-calculus, but can be described in ML or in the second order typed λ-calculus.

The type reconstruction results fully characterize the problem for ML. However, the proof techniques could be useful for studying more complex type disciplines. For example, they generalize easily to derive lower bounds on recognizing typable terms for System F, the second-order polymorphic λ-calculus, and various higher-order extensions, particularly F_ω.

In Conclusion: It is worth commenting on the orthogonality of coding methods used in deriving expressibility and type reconstruction bounds, respectively. The expressibility results depend critically on homogeneous lists, where one iterates over lists of terms with identical type. The generic simulation results of the expressibility theorems *require* that the tape symbols have identical type, in order to make the classic iterative machinery go through. By contrast, the type reconstruction bounds depend on complex data structures, including lists, over heterogeneous atomic elements. For example, a Turing machine tape in the type reconstruction proof is coded as iterated *pairs* of coded tape symbols, where the type of the coded tape reflects uniquely the tape contents.

The moral of these proofs seems to be that each new coding method reveals some more of the elegance of the simply typed λ-calculus.

References

1. S. Abiteboul and C. Beeri. *On the Power of Languages for the Manipulation of Complex Objects*. INRIA Research Report 846, 1988.
2. S. Abiteboul and P. Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *SIGACT News*, **21** (1990), pp. 9–18.
3. S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *J. Comput. System Sci.*, **43** (1991), pp. 62–124.
4. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.
5. S. Bellantoni and S. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. In *Proceedings of the 24th ACM STOC* (1992), pp. 283–293.
6. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings DBPL3*, pp. 9–19. Morgan-Kaufmann, 1991.
7. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *Proceedings 4th ICDT*, pp. 140–154. LNCS 646, Springer Verlag, 1992.
8. V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *Proceedings of the 18th ICALP*, pp. 60–75. LNCS 510, Springer Verlag, 1991.
9. A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *J. Comput. System Sci.*, **25** (1982), pp. 99–128.
10. A. Chandra and D. Harel. Horn Clause Queries and Generalizations. *J. Logic Programming*, **2** (1985), pp. 1–15.
11. A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.
12. 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.
13. E. Codd. Relational Completeness of Database Sublanguages. In R. Rustin, editor, *Database Systems*, pp. 65–98. Prentice Hall, 1972.
14. L. Damas and R. Milner. Principal Type Schemes for Functional Programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, January 1982.
15. C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the Sequential Nature of Unification. *Journal of Logic Programming* **1** (1984), pp. 35–50.
16. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. *SIAM-AMS Proceedings*, **7** (1974), pp. 43–73.
17. 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.
18. J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur*. Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

19. J.-Y. Girard, A. Scedrov, P.J. Scott. Bounded Linear Logic: a Modular Approach to Polynomial Time Computatability. In *Feasible Mathematics*, ed. Samuel R. Buss and Philip J. Scott, Birkhauser 1990, pp. 195-210.
20. Y. Gurevich. Algebras of Feasible Functions. In *Proc. of the 24th IEEE FOCS* (1983), pp. 210–214.
21. R. Harper, R. Milner, M. Tofte. *The Definition of Standard ML*. MIT Press, 1990.
22. F. Henglein and H. G. Mairson. The Complexity of Type Inference for Higher-Order Typed Lambda Calculi. In *Proceedings of the 18th ACM Symposium on the Principles of Programming Languages*, pp. 119–130, January 1991.
23. G. Hillebrand, P. Kanellakis, and H. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. In *Proceedings of the 8th IEEE LICS* (1993), pp. 332–343.
24. G. Hillebrand, P. Kanellakis. Functional Database Query Languages as Typed Lambda Calculi of Fixed Order. In *Proceedings of the 13th ACM PODS* (1994).
25. G. Hillebrand. Finite Model Theory in the Simply Typed Lambda Calculus. *PhD Thesis* Brown Univ. (1994).
26. R. Hull and J. Su. On the Expressive Power of Database Queries with Intermediate Types. *J. Comput. System Sci.*, **43** (1991), pp. 219–267.
27. N. Immerman. Relational Queries Computable in PTIME. *Info. and Comp.*, **68** (1986), pp. 86–104.
28. N. Immerman, S. Patnaik, and D. Stemple. The Expressiveness of a Family of Finite Set Languages. In *Proceedings of the 10th ACM PODS* (1991), pp. 37–52.
29. P. Kanellakis, H. Mairson, and J. Mitchell. Unification and ML-type Reconstruction. In *Computational Logic: Essays in Honor of Alan Robinson*, pp. 444–478. MIT Press, 1991. (Final version of: P. C. Kanellakis and J. C. Mitchell. Polymorphic Unification and ML Typing. In *Proceedings of the 16th ACM Symposium on the Principles of Programming Languages*, pp. 105–115, January 1989. H. G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the 17th ACM Symposium on the Principles of Programming Languages*, pp. 382–401, January 1990.)
30. A. Kfoury, J. Tiuryn, and P. Urzyczyn. An Analysis of ML Typability. In *Proceedings 17th Colloquium on Trees, Algebra and Programming*, pp. 206–220. LNCS 431, Springer Verlag, 1990.
31. A. Kfoury, J. Tiuryn, and P. Urzyczyn. The Hierarchy of Finitely Typed Functional Programs. In *Proceedings 2nd IEEE LICS* (1987), pp. 225–235.
32. P. Kolaitis and C. Papadimitriou. Why Not Negation By Fixpoint? *J. Comput. System Sci.*, **43** (1991), pp. 125–144.
33. G. Kuper and M. Vardi. On the Complexity of Queries in the Logical Data Model. *Theoretical Comput. Sci.*, **116** (1993), pp. 33–57.
34. R. E. Ladner. The Circuit Value Problem is Logspace Complete for P. *SIGACT News* **7:1**, 1975, pp. 18–20.
35. D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. In *Proc. of the Inter. Conf. on Typed Lambda Calculi and Applications*, Utrecht 1993. (To appear in *Fundamenta Informaticae*.)
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.G. Mairson. A Simple Proof of a Theorem of Statman. *TCS*, **103** (1992), pp. 387–394.

38. H. G. Mairson. Quantifier Elimination and Parametric Polymorphism in Programming Languages. *J. Functional Programming* bf 2:2 (April 1992), pp. 213–226.
39. A. R. Meyer. The Inherent Computational Complexity of Theories of Ordered Sets. *Proceedings of the International Congress of Mathematicians*, 1974, pp. 477–482.
40. R. Milner. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.*, **17** (1978), pp. 348–375.
41. C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.
42. J. C. Reynolds. Towards a Theory of Type Structure. In *Proceedings of the Paris Colloquium on Programming*, Lecture Notes in Computer Science 19, Springer Verlag, pp. 408–425, 1974.
43. J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM* **12:1** (1965), pp. 23–41.
44. H. Schwichtenberg. Definierbare Funktionen im λ-Kalkül mit Typen. *Archiv für mathematische Logik und Grundlagenforschung*, **17** (1976), pp. 113–114.
45. R. Statman. The Typed λ-Calculus is not Elementary Recursive. *Theoretical Computer Sci.*, **9** (1979), pp. 73–81.
46. W. W. Tait. Intensional Interpretation of Functionals of Finite Type I. *J. Symbolic Logic* **32** (1967) pp. 198–212.
47. M.Y. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM STOC* (1982), pp. 137–146.
48. M. Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae* **10** (1987).