The Complexity of Uniform Traversal Combinator

Sushant Patnaik

CMPSCI Technical Report 96-26
April, 1996

Abstract

In [FSS92], Fegaras, Sheard and Stemple propose and study a family of algebraic database programming languages based on a new recursion scheme, the so-called Uniform Traversal Combinator (UTC), that greatly facilitates verification and theorem proving. The complexity of their languages was left as an open question. It turns out that their novel recursion construct imposes a syntactic restriction that is akin to the formulation independently conceived of by Bellantoni and Cook (in [BC92, B92]), where they separate *safe and normal* variables in the notational (and predicative) primitive recursion template and they prove that the resulting recursion schemes capture exactly the functions in P and Linear Space. We combine the two perspectives to show how the presence of *types* in association with this restricted form of recursion limits the complexity of the resulting languages. In particular, we use the proof techniques of [BC92] to show that the class of functions, from $\mathcal{N} \rightarrow \mathcal{N}$, that can be expressed in the UTC-languages are exactly the functions in the complexity classes: Linear Space, P, Logspace and ETIME, depending on whether traversal on the typed objects, *integer* or *list* of *integer* or an appropriate combination, is allowed.

Keywords: Semantics of programming languages, expressive complexity, types, database transaction languages, recursion theory.

1. Introduction

In [FSS92], Fegaras, Sheard and Stemple proposed a new algebraic language based on a specific traversal scheme for simulating recursion and looping that was better suited for theorem proving. Bulk data structures are defined inductively by a small number of type constructors. There is only one primitive for traversing structures: the Traversal Combinator, which is defined inductively according to the recursive type definition of the structures being traversed. However, the latter is a variant of the usual primitive recursive template and it captures all of Primitive Recursive Functions (PrimRec). They then imposed a syntatic restriction that the accumulative variables could not be traversed and they referred to it as the Uniform Traversal Combinator, UTC. Here, traversals traverse variables only, and not the results of other computations such as other traversals. This restriction limits the search space of optimization of programs making the program translation and optimization process very effective and efficient. They showed how such an algebraic framework is used to facilitate automated theorem-proving. Database transactions, when specified in this algebra, can then be checked efficiently (in an automated way) to see if they satisfy any integrity constraints or, other theorems regarding the properties of such programs can be proved systematically. Most standard optimization techniques such as pushing a selection inside a join, are captured by a single reduction method. The salient features of their algebra are as follows ([FSS92]):

- it is rich enough to capture most bulk data types,
- programs are expressed in a highly stereotyped recursive form,
- the theorem prover is just a reduction algorithm that uses the inductive properties of the program of programs,
- abstract programs are translated by the type transformation model into concrete programs that are expressed in concrete primitives only,
- cost functions for uniform traversal combinators are easier to write than for more general forms,
- integrity constraints attached to types offer alternative methods of execution.

However, its exact complexity was not known prior to this.

In [BC92, B92], Bellantoni and Cook, motivated by purely theoretical reasons, consider a syntactic restriction on primitive recursion wherein recursive terms (denoted as *safe*) are not allowed to be substituted into a position which was used for an earlier definition by recursion. They call this restricted form of primitive recursion - *Predicative Primitive Recursion.* They distinguish between *normal* and *safe* variables. They define a class A as the set of functions with *normal* inputs closed under this restricted form of recursion and a suitable composition that respects the *safeness* of variables. They also define a similar syntactic restriction on Cobham's ([Co64])notational recursion template and call it *Predicative Notational Recursion*, and they denote the corresponding class of functions as B. They prove that B equals the class of functions in P and A equals the class of functions in Linear Space. Their result is striking because they show that syntactic restrictions alone, without imposing any resource bounds, are sufficient to capture tractable classes inside PrimRec. The definitions in the two frameworks have the same flavor except for a couple of crucial differences. Firstly, composition is handled differently. While Bellantoni has to restrict composition explicitly by defining a *safe* version, Fegaras handles it implicitly in the semantics of the operator. The other important feature that distinguishes the UTC model from Bellantoni's framework is the support for *typed* variables. The language based on UTC is strongly typed. A possible way of enforcing types in Bellantoni's framework would be to add two different *sorts* of integers.

We show that in the UTC framework we can capture the functions in P and Linear Space, in a manner similar to Bellantoni's algebras, A and B, with UTC on *lists* (with *cdr* operation) and *integers*, respectively. Further, we exhibit the interesting interaction of types with the complexity. Allowing both *integer* and *list* typed variables, and taking closure under composition in the UTC framework, we capture the class of functions in ETIME, allowing UTC on *lists* and allowing only *integer* typed variables in the UTC template for storing the partial values and taking closure under composition, we capture exactly the functions in Logspace and allowing only *integer* types and extending the algebra to include two new notational successor operations, namely, $succ_1(x) = 2x + 1$ and $succ_0(x) = 2x$, as initial functions and taking closure under composition, we get the class of *Elementary Functions.*

Note that the complexity results here differ from the results of Gurevich([Gu83]), Immerman([Imm82]), Immerman etal. ([IPS91]), Vardi ([Va82]) and others in the finite model theoretic framework, where the domain over which relations are interpreted is finite. In fact, in [Gu83], Gurevich shows that primitive recursion over finite domains exactly captures the functions in Logspace. On the other hand, in [L92], Leivant reports interesting results on characterizing functions in polynomial time (FP) in a flavor similar to Bellantoni and Cook's framework. He uses a free term algebra, which is like the *list* type (list of bits representing the integer), to realise FP.

In Section 2., we formally define the Uniform Traversal Combinator and the class of languages based on it. In Section 3., we state our results on resolving the complexity of UTC based classes and give the proofs in the following section. The techniques used are based on standard complexity theoretic arguments and Bellantoni's ([BC92]) results. We are able to get a slightly different (from that of [BC92]) characterization of functions in Linear Space and a new characterization of functions in Logspace, ETIME and HYPEREXP (or, *Elementary Functions*). Finally, in Section 5., we mention some open problems in characterizing the complexity of UTC, and we pose a question about the complexity of FOLD over *integer* types.

2. Definitions

Definition 2.1 (Traversal combinator)([FSS92]) Let T be a canonical type with constructors $C_i(\overline{x_i}, \overline{y_i})$. A traversal combinator $\mathcal{H}_T(f_1, \ldots, f_n) : T \rightarrow b$, where b is any type, is defined in general, as follows: ([FSS92])

$[x] \rightarrow$ **case** x
$\quad\{\quad \ldots$
$\quad\quad C_i(\overline{x_i}, \overline{y_i}) \;\rightarrow\; f_i(\overline{x_i}, \overline{y_i}, \mathcal{H}_T(f_1, \ldots, f_n)(y_1^i), \ldots, \mathcal{H}_T(f_1, \ldots, f_n)(y_{i_r}^i));$

```
        ...
}
```

In the Traversal Combinator, the types can be arbitrary structured types. Indeed, in [FSS92], the authors investigate recursion with various data types such as lists, integers, trees, etc.. In this paper, we shall only look at traversal combinators over two types: *integer* and *list* of 0 and 1's (or, in general, integers). Thus, T = *integer* or *list*, and we denote the corresponding combinator, $\mathcal{H}_T(f_1, \ldots, f_n)$, as TC-INT$(f_1, \ldots, f_n)$ and TC-LIST$(f_1, \ldots, f_n)$, respectively.

Definition 2.2 For any positive *integer*, i, the operator is:

$$\text{TC-INT}(f,g)(i) \equiv (\mathit{zero} \rightarrow g(), \lambda(a,r)\ f(a,r))(i)$$

$$\equiv\ f(i, (f(i-1, \ldots, f(\mathit{succ}(0), g()) \ldots)))$$

For any *list*, $l = \langle l_1, \ldots, l_n \rangle$, where $l_i \in \{0,1\}$, the operator is:

$$\text{TC-LIST}(f,g)(l) \equiv (\text{nil} \rightarrow g(), \lambda(a,r)\ f(a,r))(l)$$

$$\equiv\ f(\langle l_1, \ldots, l_n \rangle, f(\langle l_2, \ldots, l_n \rangle, \ldots, f(\langle l_n \rangle, g()) \ldots))$$

The variable r is called an accumulative variable.

We define *Uniform*-TC, or UTC operation, as the above traversal but with the restriction that variable r, in the template above, or any value returned by any function, say, f, using r, cannot be recursed or traversed upon i.e. the language does not allow a TC-INT or TC-LIST, as the case may be, on any accumulative variable.

The recursion template resembles the primitive recursion template but the crucial difference arises from the new semantics. Note that this restricts composition inside the scope of an accumulative variable in a way such that the value returned by any function, that is defined in terms of a function that itself uses an accumulative variable, cannot be recursed upon. It may seem, at first glance, to be a very constraining requirement but surprisingly enough, as we shall see shortly, it can express a relatively large class of functions.

Adopting the notation in [BC92], we separate the accumulative from the normal variables in the function argument list by a semi-colon. Thus, in $f(y, x; r, c)$, variables, r, c, to the right of the semi-colon, are accumulative (or, *safe* as in [BC92]) and hence, cannot be recursed upon.

Example 2.3 Consider the following examples of TC-INT and TC-LIST operator. Let $succ(x) = x + 1$, for any integer, x. Let x, y be two positive integers.

$$\text{Add}(x; y) \equiv \text{TC-INT}(0 \rightarrow y, \lambda(a, r) succ(r))(x)$$

computes $x + y$.

$$\text{Mult}(x, y) \equiv \text{TC-INT}(0 \rightarrow 0, \lambda(a, r)\text{Add}(y; r))(x)$$

computes xy.

$$\text{Length}(l) \equiv \text{TC-LIST}(nil \rightarrow 0, \lambda(a, r) succ(r))(l)$$

computes the length of a list, l.

$$\text{Exp(x,y)} \equiv \text{TC-INT}(0 \rightarrow 1, \lambda(a, t)\text{Mult}(y, t))(x)$$

computes y^x.

Add and Mult are UTC-INT expressions and Length is an UTC-LIST expression, since they do not recurse on the accumulative variable, r. Exp is not a UTC-INT expression, since it recurses on the accumulative variable, t, when the functions, Mult and then, Add, are invoked. In fact, one cannot compute y^x using UTC-INT recursion.

Definition 2.4 Following Bellantoni, a function f is defined using *safe composition* from functions g, h, r if:

$$f(\bar{x}; \bar{a}) = g(h(\bar{x};); r(\bar{x}; \bar{a})).$$

The idea is that composition should not violate the non-recursiveness property. Hence, the values returned by functions that use accumulative variables in the TC template are ensured to remain safe i.e. non-recursive. *Safe* composition is implicitly enforced by the restriction that defines the Uniform Traversal Combinator.

Let $\mathcal{N}$ denote the class of Natural numbers. Hereafter, when we refer to a class of functions, we mean functions from $\mathcal{N} \rightarrow \mathcal{N}$. Let $\mathcal{I}$ and $\mathcal{L}$ denote *integer* and *list of integer* types respectively. Depending on whether functions in the language under consideration return values that are *list* or *integer* typed, we shall distinguish between types of function families: $\mathcal{F}_1 : \bar{\mathcal{I}} \times \bar{\mathcal{L}} \rightarrow \bar{\mathcal{I}}$, $\mathcal{F}_2 : \bar{\mathcal{I}} \times \bar{\mathcal{L}} \rightarrow \bar{\mathcal{L}}$. Abusing notation, we shall consider functions from $\bar{\mathcal{I}} \rightarrow \bar{\mathcal{I}}$ and $\bar{\mathcal{L}} \rightarrow \bar{\mathcal{L}}$, as special cases of $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively.

Definition 2.5 Let UTC(int, =) denote the smallest class of functions $\in \mathcal{F}_1$ containing

1. Zero function, 0
2. Projection(.), e.g. any tuple variable x, of width k, $x.i$, $1 \leq i \leq k$; and Tupling ([]), e.g. $\bar{y} = [x_1, \ldots, x_k]$
3. Succ e.g. $\text{succ}(i) = i + 1$
4. Equality $(x = y)$
5. Conditional (if $a = 0$ then b else c)

and closed under UTC-INT operator and *safe* composition.

Note that composition outside the scope of any accumulative variable is the usual composition.

Definition 2.6 Let UTC(*int*, *pred*) denote the corresponding class of functions with $pred(i) = i - 1$ (predecessor) replacing $=$ on integers as an initial function.

Let UTC(*int*) denote the corresponding class of functions without $=$ as an initial function.

Definition 2.7 Let UTC($int, pred, succ_0, succ_1$) be the smallest class of functions $\in \mathcal{F}_1$ containing

1. Zero function, 0
2. Projection(.), e.g. any tuple variable x, of width k, $x.i$, $1 \leq i \leq k$; and Tupling ([]), e.g. $\bar{y} = [x_1, \ldots, x_k]$
3. Succ e.g. $\text{succ}(i) = i + 1$
4. $\text{Succ}_0(i) = 2i$
5. $\text{Succ}_1(i) = 2i + 1$
6. $\text{Pred}(x) = x - 1$
7. Conditional (if $a = 0$ then b else c)

and closed under UTC-INT operator and *safe* composition.

In the following definition, we only allow lists containing 1's and 0's, and we use such lists to encode numbers in the obvious way.

Definition 2.8 Let UTC(*list*, *cdr*) denote the smallest class of functions $\in \mathcal{F}_2$ containing

1. Nil
2. Projection(.), e.g. any tuple variable x, of width k, $x.i$, $1 \leq i \leq k$; and Tupling ([]), e.g. $\bar{y} = [x_1, \ldots, x_k]$ where x_i are lists.
3. Cons e.g. $\text{Cons}(a, \langle \ldots \rangle) = \langle a, \ldots \rangle$
4. Cdr e.g. $\text{Cdr}(\langle l_1, l_2, \ldots, l_n \rangle) = \langle l_2, \ldots, l_n \rangle$
5. Conditional (if $\text{Car}(l_1) = 0$ then l_2 else l_3)

and closed under UTC-INT operation and *safe* composition.

Note that we allow equality on the elements of the list. Any function expressed in this language inputs a list-typed variable and outputs a list-typed value.

Definition 2.9 Let UTC(*list*, =) denote the corresponding class of functions as above with equality on lists replacing *cdr* as an initial function.

Let UTC(*list*) denote the corresponding class of functions as above without *cdr* as an initial function.

The following definitions differ from the previous in that the input and output values of the functions expressed in the language may be of different types.

Definition 2.10 Let UTC(*list*, *cdr*, *int*, *pred*) denote the smallest class of functions $\in \mathcal{F}_1, \mathcal{F}_2$ containing the initial functions of Definitions 2.5 and 2.8 and closed under TC-INT and TC-LIST operations and *safe* composition.

Let UTC(*list*, *cdr*, *int*, =) denote the corresponding class of functions as above with = on integers replacing *pred* on integers as an initial function.

Definition 2.11 Let UTC(*list*, *boundint*, =) (or, UTC(*list*, *boundint*, *pred*)) denote the smallest class of functions $\in \mathcal{F}_1$ containing the following initial functions on *integers*:

1. Zero function, 0
2. Projection(.), e.g. any tuple variable x, of width k, $x.i$, $1 \leq i \leq k$; and Tupling ([]), e.g. $\bar{y} = [x_1, \ldots, x_k]$

3. Succ e.g. $\mathrm{succ}(i) = i + 1$

4. Equality $(x = y)$ (or, $\mathrm{Pred}(x) = x - 1$)

5. Conditional (if $a = 0$ then b else c)

and closed under TC-LIST operation and *safe* composition.

We constrain the algebra to accept inputs of *list* type and return *integer* typed values. This constraint on types is somewhat contrived, but it serves to capture logspace, as we shall see shortly.

Let n be the input size ($= \log x$, if x is an input integer). Let FL denote the class of functions computable in logspace i.e. SPACE$[\log n]$.
Let FP denote the class of functions in polynomial time i.e. TIME$[n^{O(1)}]$.
Let FLS denote the functions in Linear Space i.e. SPACE$[n]$.
Let FETIME denote the functions computed by deterministic Turing machines in time $O(2^{O(n)})$. Note that FLS $\subseteq$ FETIME.

Let FHYPEREXP denote the functions computable in time $O(2^{2^{\cdot^{\cdot^{\cdot^{2^{x^k}}}}}})$, where the height of the tower of 2's is $O(1)$.

Definition 2.12 *([R])* The *Elementary functions* are the smallest class of functions containing zero, succ, addition, multiplication, projection and closed under bounded addition and bounded multiplication and composition.

Definition 2.13 *([R])* Let $\mathcal{E}^3$ denote the 3rd level of the Gregorzyck hierarchy which is defined to be the smallest class of functions containing the functions: zero, succ, projection, E_0, E_1 and E_2:

$$E_0(x,y) = x + y$$
$$E_1(x) = x^2 + 2$$
$$E_2(x+1) = E_1(E_2(x))$$

and closed under composition and *limited recursion*: a function f is defined by *limited recursion* from g, h, j

$$\begin{aligned} f(0,y) &= g(y) \\ f(x+1,y) &= h(x,y,f(x,y)) \\ f(x,y) &\leq j(x,y) \end{aligned}$$

$\mathcal{E}^3$ is known to be equal to the class of Elementary Functions, a proper subclass of PrimRec ([R]) (since the Gregorzyck hierarchy is a strict hierarchy and PrimRec = $\bigcup_i \mathcal{E}^i$). It is well known that

Fact 2.14 $\mathcal{E}^3$ = FHYPEREXP.

3. Results

We characterize the exact complexity of the UTC classes defined in the previous section. The *types* allowed in the language restrict the complexity of the functions that can be expressed. UTC(*list*, *cdr*) is functionally similar to Bellantoni's B algebra, but couched in a more algebraic language framework. For completeness sake, we show here that it can express exactly the polytime functions. In one direction our proof is different from his.

Theorem 3.1 *([BC92]) UTC(list, cdr) = FP.*

For integer types, adding *pred* or = does not make a difference to expressiveness. Bellantoni's C algebra is identical to UTC(*int*, *pred*), except for the differences in the frameworks, mentioned earlier. His proof basically shows that UTC(*int*, *pred*) = FLS. Replacing *pred* with =, we can capture the same class.

Theorem 3.2 *UTC(int, =) = FLS.*

Since we have *integer* typed variables and *succ* and = on integers in UTC(*list*, *boundint*, =), we can capture Logspace exactly. Bellantoni captures Logspace by restricting the output to be at most logarithmic size. As it turns out, constraining the values of each function to be integer typed in the TC-LIST template achieves the same purpose. Capturing logspace in a more natural way in this framework seems difficult.

Theorem 3.3 *UTC(list, boundint, =) = FL.*

Functions in ETIME are in general not closed under composition. But in the framework of UTC, the *safe* composition scheme maintains each *integer* value to be of linear length and each *list* to be of exponential length. Thus, in a language framework with support for a *typing* scheme, we have a "resource-independent characterization" (in the terminology of [BC92]) of functions in ETIME. We do not constrain the recursion by explicitly bounding the values of the functions or variables. Indeed, typing constraints seem to be more natural than limiting recursion depth explicitly (as in the seminal result of [Co64]), especially in the context of programming languages.

Theorem 3.4 *$UTC(list, cdr, int, =) = FETIME$.*

UTC($int, pred, succ_0, succ_1$) is much more expressive, since, we can recurse on any integer using either Predicative Notational or Predicative Primitive recursion or both and hence we cannot bound the lengths of any integer tractably. However, it is still strictly contained in PrimRec.

Proposition 3.5 *$UTC(int, pred, succ_0, succ_1) = \mathcal{E}^3$ (= FHYPEREXP).*

4. Proofs

One direction of all our proofs relies on a technical lemma (4.2) which for different cases gives upper bounds on the size of any function expressed in the different UTC languages. We prove Theorem 3.4 first because it uses the most general version of the above mentioned lemma. The other theorems use various restricted versions of it. The proof of the lemma is an adaptation of Bellantoni's result ([BC92, B92]). We shall use the following notation:
For any (tuple of) positive *integers*, $\bar{x}$, let $l(\bar{x}) = \bar{x}$ and $|\bar{x}|$ (or, $lg(\bar{x})$) denote the vector of lengths of the binary representation of each integer, x_i i. e. $\lceil \log_2 x_1 \rceil, \ldots, \lceil \log_2 x_c \rceil$.
For any *lists*, $\bar{x}$, let $l(\bar{x}) = |x_1|, \ldots, |x_c|$, where $|x_i|$ is the length of list x_i.

Proof: (Theorem 3.4)
To show that FETIME $\subseteq$ UTC($int, pred, list, cdr$), let M be any ETIME Turing machine that on input x computes $f(x)$. Let δ_M denote its transition function, let its alphabet $\Sigma = \{0, 1\}$, let Q $= \{q_0, \ldots, q_c\}$ be the state set. Let $|y|$ denote the length of y if y is of type *list*, else $lg(y)$ if y is of type *integer*. By assumption, there exists a constant k, such that M computes $f(x)$ in number of steps bounded by $2^{k|x|}$). Let W denote the input integer (or, list). Let y be any integer, and let EXP(y) return an *integer* of value $2^{k|y|} = y^k$, for some constant k. Let BIT(p, y) denote the pth bit of y. Theorem 3.2 shows that UTC($int, pred$) expresses exactly FLS. Hence, all the usual arithmetic operations can be done in FLS. In particular,

Proposition 4.1 *Given integers x, p, $EXP(x)$, $BIT(p, x)$ can be expressed in $UTC(int, pred)$.*

Proof: Let W be the input integer. Let EXP(W) $= W^k$ return an *integer*, for some constant k. Since W is the input, it is a normal variable and hence, can be traversed. EXP is thereby easily expressed using composition on a multiplication function k times:

$$\text{Mul}(x, W) \equiv \text{TC-INT}(0 \to 0, \lambda(i, r)\text{Add}(W, r))(x)$$

$$\text{EXP}(W) \equiv \text{Mul}(W, \text{Mul}(W, \ldots, \text{Mul}(W, W)) \ldots)$$

Let $\text{P2DIV}(p, x) = x/2^p$, and $\text{SUB}(y, x) = x - y$. Then, $\text{BIT}(p, x) = \text{P2DIV}(p, x)$ mod 2.

$$\text{P2DIV}(p, x) \equiv \text{TC-INT}(0 \rightarrow x, (a, r) \rightarrow r \text{ div } 2)(p)$$

$$\text{SUB}(y, x) \equiv \text{TC-INT}(0 \rightarrow x, (a, r) \rightarrow \mathit{pred}(r))(y)$$

```
$x$ div 2 $\equiv$ TC-INT($0 \rightarrow 0$,
        $(a, r) \rightarrow$ if $SUB(a + a, x) = 0$ then $a$
                   else if SUB$(a + a + 1, x) = 0$ then $a$
                   else $r$)(EXP($W$))
```

∎

Next we simulate M by using variables: L to represent the contents of the tape to the left of the head, R to represent the contents to the right, h to represent the content under the tape head, Q the state of the machine. Let INIT(W) return a *list* of length W containing each bit BIT(p, W) of the *integer* W. (If W is of type *list*, then INIT(W) = W.)

$$\text{INIT}(W) \equiv \text{TC-INT}(0 \rightarrow \mathit{nil}, \lambda(a, l)\text{cons}(BIT(a, W), l))(W)$$

We can now express M's computation as:

```
TC-INT($0 \rightarrow (\mathit{nil}, W \text{mod } 2, \mathit{cdr}(\text{INIT}(W)), q_0)$,
      $\lambda(a, L, h, R, Q)$ if $\delta_M(Q, h, q', \ , \text{L})$
                  then $(\mathit{cdr}(L), \mathit{car}(L), \mathit{cons}(h, R), q')$
                  $\vdots$
                  else if $\delta_M(Q, h, q', \ , \text{R})$
                  then $(\mathit{cons}(h, L), \mathit{car}(R), \mathit{cdr}(R), q')$
                  $\vdots$
                  else if $\delta_M(Q, h, q', \sigma, \text{S})$
                  then $(L, \sigma, R, q')$
                  $\vdots$
      ) (EXP($W$))
```

The IF-THEN-ELSE's are a finite constant sized block of statements depending on δ_M.

In the other direction, we have to show that if $f(\bar{x})$ denotes a function in this language, that returns either a *list* or an *integer*, where $\bar{x}$ are normal (i.e. they can be recursed upon) input variables (integers or lists), then $l(f(\bar{x})) \leq p_f(l(\bar{x}))$, where p_f is a polynomial. In other words, we have to show that the lengths of any *list* and any *integer* constructed in this language are exponential and linear respectively in the length (in binary) of the input integers or lists. Following Bellantoni, we bound the size of the functions computed in this class of languages. For any function $f \in \mathcal{F}_1$ or $\mathcal{F}_2$, we shall refer to $l(f())$ as the size of f. The proofs are similar for all the cases, the only difference being the proofs of the induction basis depending on whether the functions are in $\mathcal{F}_1$ or $\mathcal{F}_2$ i.e. they return integers or lists. Let p_g, p_h be two polynomials.

Lemma 4.2 $l(f(y,\bar{x};\bar{c})) \leq l(y)p_h(l(y),l(\bar{x})) + p_g(l(\bar{x})) + max_i(l(c_i)+1)$.

Proof: There are several cases depending on whether $f \in \mathcal{F}_1, \mathcal{F}_2$, and whether f is defined using TC-INT or TC-LIST. (Note that fixing f's type implies that g, h are of the same type.)

$$f(y,\bar{x};\bar{c}) = \text{TC-INT}(0 \rightarrow g(\bar{x};\bar{c}), \lambda(i,r)h(i,\bar{x};r,\bar{c}))(y).$$

Case 1: When $f \in \mathcal{F}_1$ i.e. $f : \bar{\mathcal{I}} \times \bar{\mathcal{L}} \rightarrow \mathcal{I}$, and y is *integer* typed. Case 2: When $f \in \mathcal{F}_2$ i.e. $f : \bar{\mathcal{I}} \times \bar{\mathcal{L}} \rightarrow \mathcal{L}$, and y is *integer* typed.

$$f(y,\bar{x};\bar{c}) = \text{TC-LIST}(0 \rightarrow g(\bar{x};\bar{c}), \lambda(i,r)h(i,\bar{x};r,\bar{c}))(y).$$

Case 3: When y is of type *list* and $f \in \mathcal{F}_1$. Case 4: When y is of type *list* and $f \in \mathcal{F}_2$.

The proof is by induction on the depth of derivation of f. The proofs for Cases 3, 4 are analogous to those for 1, 2.

Basis: The initial functions (for the case 1 when they return integers as in Definition 2.5, or lists for case 2 as in Definition 2.8) are easily seen to satisfy the assertion.
$l(y) = 0$: By definition of f, we see that $l(f(y,\bar{x};\bar{c})) = l(g(\bar{x};\bar{c})) \leq p_g(l(\bar{x})) + \max_i(l(c_i)+1)$ (by the ind. hyp. on g)
Induction Step: Assume true for y. Consider y' such that $l(y') = l(y)+1$:

$$\begin{aligned}
l(f(y',\bar{x};\bar{c})) &= l(h(y',\bar{x};\bar{c},f(y,\bar{x};\bar{c}))) \\
&\leq p_h(l(y'),l(\bar{x})) + \max(l(f(y,\bar{x};\bar{c}))+1, l(c_i)+1) \\
&\qquad \text{(by the ind. hypothesis on } h) \\
&\leq p_h(l(y'),l(\bar{x})) + l(y)p_h(l(y),l(\bar{x})) + 1 + p_g(l(\bar{x})) + \max_i(l(c_i)+1) \\
&\qquad \text{(by the ind. hypothesis on } f) \\
&\leq l(y')p_h(l(y'),l(\bar{x})) + p_g(l(\bar{x})) + \max_i(l(c_i)+1).
\end{aligned}$$

That completes the induction step. ∎

For *safe* composition, let

$$f(\bar{x};\bar{c}) = h(\bar{g}(\bar{x}); \bar{r}(\bar{x};\bar{c}))$$

Lemma 4.3 $l(f(\bar{x};\bar{c})) \leq p_f(l(\bar{x})) + \max_i(c_i+1)$, where p_f is some polynomial.

Proof: We induct on the composition depth. By the induction hypothesis on g, h, r, we have corresponding polynomials p_{g_i}, p_h, p_{r_i}, respectively, such that $l(h(\bar{x};\bar{c})) \leq p_h(l(\bar{x})) + \max_i(l(c_i)+1)$, $l(g_i(\bar{x})) \leq p_{g_i}(l(\bar{x}))$ and $l(r_i(\bar{x};\bar{c})) \leq p_{r_i}(l(\bar{x})) + \max_i(l(c_i)+1)$.

$$\begin{aligned} l(f(\bar{x};\bar{c})) &= l(h(\bar{g}(\bar{x}); \bar{r}(\bar{x};\bar{c}))) \\ &\leq p_h(l(\bar{g}(\bar{x}))) + \max_i(p_{r_i}(l(\bar{x})) + \max_j(c_j+1)+1) \\ &\leq p_h(\bar{p_g}(l(\bar{x}))) + p_r(l(\bar{x})) + 1 + \max_i(c_i+1) \\ &\leq p_f(l(\bar{x})) + \max_i(c_i+1) \end{aligned}$$

where $p_f(l(\bar{x})) = p_h(\bar{p_g}(l(\bar{x}))) + p_r(l(\bar{x})) + 1$. That completes the induction step. ∎

For example, in the particular case of $f \in \mathcal{F}_2$, and $y, \bar{x}, \bar{c}$ are integers, we have that

$$|f(y,\bar{x};\bar{c})| \leq y p_h(y,\bar{x}) + p_g(\bar{x}) + \max_i(c_i+1).$$

The proof is completed by noting that all input variables are normal, i.e. they can be recursed upon. Therefore, $l(\bar{c})$, for the accumulative variables, $\bar{c}$, whether lists or integers, is bounded by a polynomial in $l(\bar{x})$ where, $\bar{x}$ denote the input variables (and hence, by an exponential in the lengths (in binary representation) of the *integer* typed variables, since for any accumulative *integer* variable, c, and input integer, w, $l(c) \leq l(w)^k \rightarrow c \leq w^k = 2^{k(lg(w))}$.). By Lemmae 4.2,4.3, the length of any output *lists* computed by any function in this language is polynomial in the values of the *integer* variables and polynomial in the lengths of the *list* variables. Thus, we have that

Lemma 4.4 $l(f(\bar{x})) \leq p_f(l(\bar{x}))$, where p_f is a fixed polynomial.

If any function returns *integer* values, they can be written in binary using space at most linear in the (binary) lengths of the input integers, and if it returns *list* values, they have lengths exponential in the (binary) length of the input integers, and hence, can be written using time exponential in the input length. The initial functions are easily seen to be in ETIME. Let g, h be functions in ETIME. Let f be defined from g, h using TC-LIST as follows (The proof for TC-INT case is analogous.) :

$$f(y, \bar{x}; \bar{a}) = \text{TC-LIST}(nil \rightarrow g(\bar{x}; \bar{a}), \lambda(i, r)h(i, \bar{x}; r, \bar{a}))(y)$$

A Turing Machine can easily compute f by setting up a counter to run through variable y, an integer (of size at most linear in the input size) or list (of length at most exponential in the input size), and compute g and h as it goes along, needing at most time $l(y)(l(y) + l(\bar{x}) + l(\bar{c}) + \max(l(g()), l(h())))$ to store and update the partial result(s). By the inductive hypothesis, $l(y), l(\bar{c}), l(g()), l(h())$ are exponential in the length (in binary) of the input intgers. So, the time taken by the Turing machine is order of exponential in the size of its input. Similarly, if f is defined using *safe* composition from functions g, h, r, then as shown before, since $l(f())$ is at most exponential in the size of the inputs, a Turing machine needs only exponential time to compute f. Likewise, the case, when f is defined using composition from g, h, r, can be handled similarly. ∎

Proof: (Theorem 3.1)
UTC(*list*, *cdr*) $\subseteq$ FP follows by applying Lemmae 4.2,4.3 to the case when f, g, h are in $\mathcal{F}_2$ and all input and output values and all variables are *list* typed. Noting that all input variables are normal i.e. they can be traversed upon by a TC-LIST recursion, we conclude that the length of each list computed by any function, say, f in this language is bounded by a polynomial in the length of the input variables (lists) and the length of the accumulative variables (lists). By induction, length of each accumulative variable is itself bounded by a polynomial in the length of the input. The initial functions of the algebra are clearly in FP. Let g, h be functions in FP. Let

$$f(y, \bar{x}; \bar{a}) = \text{TC-LIST}(nil \rightarrow g(\bar{x}; \bar{a}), \lambda(i, r)h(i, \bar{x}; r, \bar{a}))(y)$$

Since g and h require only poly time for their computation, a Turing Machine can easily compute f by running through the polynomially long list y and computing g and h as it goes along, needing time at most some polynomial in $(|y| + |\bar{x}| + |\bar{a}| + \max(|g()|, |h()|))$ to manipulate the partial result. By the inductive hypothesis, lengths of $\bar{a}, g, h$ are polynomial in the length of the input.

In the other direction, let M be any polytime Turing machine that on input x computes $f(x)$. Let its alphabet $\Sigma = \{0, 1\}$, let Q $= \{q_0, \ldots, q_c\}$ be the state set. Let δ_M denote the usual transition function that encodes whether M on reading the bit under its head in a given state changes state and moves right or left or writes a new bit and remains stationary.

Let $|y|$ denote the length of y. By assumption, there exists a constant k, such that M computes $f(x)$ in number of steps $\leq O(|x|^k)$. Let y be a list and let $poly(y)$ return a list of lenght $|y|^k$.

Proposition 4.5 *Given any list typed variable x, $poly(x)$ can be expressed in UTC(list).*

Proof:

$$Add(x,y) \equiv \text{TC-LIST}(nil \rightarrow y, \lambda(a,r)Cons(a,r))(x)$$

computes a list of length $|x| + |y|$.

$$Square(x) \equiv \text{TC-LIST}(nil \rightarrow nil, \lambda(a,r)Add(x,r))(x)$$

computes a list of length $|x|^2$. Now $|x|^k$ can be obtained by composing k times. ∎

Next we simulate M on input $w = \langle w_1 w_2 \ldots w_n \rangle$ by using variables, L to represent the contents of the tape to the left of the head, R to represent the contents to the right, h to represent the content under the tape head, and Q the state of the machine. Let $\bar{w}$ denote the input list. We can express M's computation by:

```
TC-LIST($nil \rightarrow (nil, w_1, \langle w_2, \ldots, w_n \rangle, q_0)$,
        $\lambda(a, L, h, R, Q)$ if $\delta_M(Q, h, q',\ , \text{L})$
                         then $(cdr(L), car(L), cons(h, R), q')$
                         $\vdots$
                         else if $\delta_M(Q, h, q',\ , \text{R})$
                         then $(cons(h, L), car(R), cdr(R), q')$
                         $\vdots$
                         else if $\delta_M(Q, h, q', \sigma, \text{S})$
                         then $(L, \sigma, R, q')$
                         $\vdots$
        ) $(poly(\bar{w}))$
```

The IF-THEN-ELSE's are a finite constant sized block of statements depending on δ_M. ∎

Proof: (Theorem 3.2)
UTC($int, =$) $\subseteq$ FLS follows from application of Lemmae 4.2, 4.3 once we note that $f, g, h \in \mathcal{F}_1$ and all the input and output values and all variables are *integer* typed and hence, the value of any function, say, $f(y, \bar{x}; \bar{a})$ computed in this language is polynomial in the values of the normal variables, $y, \bar{x}$, and that of accumulative variables, $\bar{a}$.

All the input integers are normal and by induction, each of the accumulative variables have values that are polynomial in the value of the inputs. The initial functions of the algebra are clearly in FLS. Let g, h be functions in FLS. Let

$$f(y,\bar{x};\bar{a}) = \text{TC-INT}(0 \rightarrow g(\bar{x};\bar{a}), \lambda(i,r)h(i,\bar{x};r,\bar{a}))(y)$$

We have to show that f is in FLS. Since g and h require only linear space for their computation, a Turing Machine can easily compute f by cycling through all values of y and computing g and h as it goes along, needing space at most linear in $(lg(y) + lg(\bar{x}) + lg(\bar{a}) + \max(lg(g()), lg(h())))$ to store the partial results. By induction, values of $\bar{a}, g, h$ are polynomial in the values of the input integers, and hence, the length of their binary representation is linearly related to the length of the inputs. The case when f is defined by composition from g, h, r can be handled similarly.

In the other direction, let M be any linear space Turing machine that on input x computes $f(x)$. Let δ_M denote its transition function, let its alphabet $\Sigma = \{0,1\}$, let $Q = \{q_0, \ldots, q_c\}$ be the state set. By assumption, there exists a constant k, such that M computes $f(x)$ using space at most $klg(x)$, and hence, in number of steps bounded by $O(x^k)$. Let y be any integer and let $poly(y) = y^k$. Similar to *list* typed variables, it can be shown that,

Proposition 4.6 *Given any integer x, $poly(x) = x^k$ can be expressed in UTC(int).*

We need a few other easy technical propositions.

Proposition 4.7 *Given any integer variable x and input w, x mod 2, x div 2, $2x + 1$, $2x$ (only values mod w^k are required) can be expressed in UTC(int, =) without recursing on x.*

Proof: x div 2 (and similarly, x mod 2) can be expressed as

```
TC-INT(0 → 0, λ(a,r)IF Add(a,a) = x ∨ succ(Add(a,a)) = x
                    THEN a
                    ELSE r)(poly(w))
```

$2x$ (and similarly, $2x + 1$) can be expressed as

```
TC-INT(0 → 0, λ(a,r)IF(adiv 2) = x ∧ (amod 2) = 0
                    THEN a
                    ELSE r)(poly(w)) ∎
```

As before, next we simulate M by using variables: L to represent the contents of the tape to the left of the head, R to represent the contents to the right, h to represent the content under the tape head, Q the state of the machine. Let W denote the input integer. We can express M's computation by:

```
TC-INT(zero → $(0, W\text{mod } 2, W\text{div } 2, q_0)$,
        $\lambda(a, L, h, R, Q)$ if $\delta_M(Q, h, q', \ , \mathrm{L})$
                                  then $(L\text{div } 2, L\text{mod } 2, 2R + h, q')$
                                  ⋮
                                  else if $\delta_M(Q, h, q', \ , \mathrm{R})$
                                  then $(2L + h, R\text{mod } 2, R\text{div } 2, q')$
                                  ⋮
                                  else if $\delta_M(Q, h, q', \sigma, \mathrm{S})$
                                  then $(L, \sigma, R, q')$
                                  ⋮
        ) (poly(W))
```

The IF-THEN-ELSE's are a finite constant sized block of statements depending on δ_M. ∎

Proof: (Theorem 3.3)
To show that UTC(*list*, *boundint*, $=$) $\subseteq$ FL, we apply Lemma 4.2, 4.3, keeping in mind that in this case the functions are in $\mathcal{F}_1$ and variables could be either of *list* or *integer* type. The *integer* typed variables cannot be recursed upon, since the language does not allow TC-INT operator. We can assume that all input variables are *lists* since, any integer input variable can only be used as a counter under bounded number of compositions. The *list* typed accumulative variables, if any, cannot be modified, since no initial functions on *lists* are allowed in the language. Hence, we may assume that all accumulative variables are *integer* typed. Thus, the value of any function, say, $f(y, \bar{x}; \bar{a})$, in this language is bounded by a polynomial in the lengths of the input *lists*, $|y|$, $|\bar{x}|$, and the values of the integer accumulative variables, $\bar{a}$. By induction, the value of each accumulative variable is a polynomial in the length of the inputs, since only the inputs (which are of type *list*) can be recursed upon. The initial functions (on integers) are clearly in FL. Let g, h be functions in FL. Let

$$f(y, \bar{x}; \bar{a}) = \text{TC-LIST}(nil \to g(\bar{x}; \bar{a}), \lambda(i, r)h(i, \bar{x}; r, \bar{a}))(y)$$

Note that g, h return *integer* typed values. Each a_i is itself bounded by a polynomial in the length of the input. Since $g()$ and $h()$ can be at most polynomial in the length of the input lists, they can be written in binary using space at most logarithmic in the lengths of the input lists. A Turing Machine can easily compute f by setting up a counter to run through the input list y needing at most space logarithmic in $(|y|+|\bar{x}|+\bar{a}+\max(g(), h()))$ to store the partial result(s). By the inductive hypothesis, $\bar{a}, g(), h()$ are polynomial in the length of the input. The proof is completed by noting that logspace is closed with respect to composition.

In the other direction, let f be any function in FL. Let M be any Turing machine and let W denote the input (of *list* type). M computes $f(W)$ using space $O(\log(|W|))$. Let c be some integer constant such that $f(W) \leq |W|^c$. Let δ_M denote its transition function, let its alphabet $\Sigma = \{0,1\}$, let $Q = \{q_0, \ldots, q_c\}$ be the state set. We are considering functions from $\mathcal{N}$ to $\mathcal{N}$ and *lists* can encode any natural number the obvious way.

For a fixed integer constant, $c > 0$, and a fixed UTC(*list*, *boundint*, $=$) expression, say P, we can express a function *Polyiterate*(W) that on input a list, W, iterates P $|W|^c$ times. Note that any function in this language, such as P, returns only *integer* values.

Proposition 4.8 *Polyiterate*(W) *is expressible in* UTC(*list*, *boundint*, $=$).

Proof:

$$\begin{array}{ll}
Polyiterate(W) \equiv \mathrm{ITERATE}_c(W) & \\
\mathrm{ITERATE}_c(W) & \equiv \text{TC-LIST}(nil \rightarrow 0, \lambda(a,R)\mathrm{ITERATE}_{c-1}(W))(W) \\
\mathrm{ITERATE}_{c-1}(W) & \equiv \text{TC-LIST}(nil \rightarrow 0, \lambda(a,R)\mathrm{ITERATE}_{c-2}(W))(W) \\
 & \vdots \\
\mathrm{ITERATE}_1(W) & \equiv \text{TC-LIST}(nil \rightarrow 0, \lambda(a,R)P(a,R,W))(W) \blacksquare
\end{array}$$

Note that the initialization part in $\mathrm{ITERATE}_1$ may vary depending on P, and the latter returns an integer. We shall abbreviate the above procedure as $\mathrm{POLYITERATE}_{P,c}(W)$.

Further, since we have $=$ and *succ* on integers, we can do mod $|W|^c$ arithmetic easily. Let $\mathrm{BIT}(p, W)$ represent the pth symbol of the list W.

Proposition 4.9 $2x$, x div 2, $x-1$, $\mathrm{BIT}(p,W)$ *can be expressed in* UTC(*list*, *boundint*, $=$).

Proof:

$$x - 1 \equiv \Pi^1(POLYITERATE_{P,c}(W))$$

where the initialization is $nil \rightarrow [0,0]$, and P is the UTC expression:

$$\lambda(a,[r1,r2])\text{IF } succ(r2) = x \text{ THEN } [r2, succ(r2)] \text{ ELSE } [r1, succ(r2)]$$

$$2x \equiv \Pi^{11}(\text{POLYITERATE}_{Q,c}(W))$$

where the initialization is $nil \rightarrow [0,0]$, and Q is the UTC expression:

```
$\lambda(a,[r1,r2])$IF $r2 = x$THEN $[succ(0), succ(r2)]$
                ELSE IF $r1 = x$ THEN $[r2, 0]$
                ELSE $[r1, succ(r2)]$
```

$$x \text{ div } 2 \equiv \text{POLYITERATE}_{R,c}(W)$$

where the initialization is $nil \rightarrow 0$, and R is the UTC expression:

$$\lambda(a,r)\text{IF } 2a = x \text{ THEN } a \text{ ELSE } r$$

```
BIT$(p, W) \equiv$ TC-LIST$(nil \rightarrow [0,0]$,
                    $\lambda(a, r1, r2)$IF $r1 = p$ THEN $[succ(r1), a]$
                    ELSE $[succ(r1), r2])(W)$ ∎
```

We can then simulate M: we use variables L to represent the contents of the worktape to the left of the head, R to represent the contents to the right, h to represent the content under the tape head, P to represent the position of the input tape head, and Q the state of the machine. We assume w.l.o.g. that the machine has one work-tape and a read-only input tape, and accordingly, $\delta_M : Q \times \Sigma \times \Sigma \times Q \times \Sigma \times \{1,-1,0\} \times \{1,-1,0\} \rightarrow \{0,1\}$ (The transition function codes the movement of the machine's input-tape head and the work-tape head separately.). We simulate M's computation as:

$$\text{POLYITERATE}_{P,c}(W)$$

where the initialization is $nil \rightarrow [0,0,0,W,succ(0),q_0]$ representing, respectively, the initial values of the worktape-contents to the left of the head's position, the current position of the head, the value of the worktape contents to the right of the head, the input, the position of the input head and the initial state; and P is the following UTC expression:

[1]Project the first coordinate

$\lambda(a, [L, h, R, w, P, Q])$ if $\delta_M(Q, h, \mathrm{BIT}(P, w), q', \ , a_1, -1)$
then $[L \text{ div } 2, L \bmod 2, 2R + h, w, P + a_1, q']$

$\vdots$

else if $\delta_M(Q, h, \mathrm{BIT}(P, w), q', \ , b_1, 1)$
then $[2L + h, R \bmod 2, R \text{ div } 2, w, P + b_1, q']$

$\vdots$

else if $\delta_M(Q, h, \mathrm{BIT}(P, w), q', \sigma, c_1, 0)$
then $[L, \sigma, R, w, P + c_1, q']$

$\vdots$

)

where, $a_1, b_1, c_1 \in \{0, -1, 1\}$ and represent directions of the input tape head movements. The IF-THEN-ELSE's are a finite constant sized block of statements depending on the transition function, δ_M. ∎

Proof: (Theorem 3.5) To show that FHYPEREXP $\subseteq$ UTC(*int, pred, succ*$_0$, *succ*$_1$), let M be any HYPEREXP time Turing machine that on input x computes $f(x)$. Let δ_M denote its transition function, let its alphabet $\Sigma = \{0, 1\}$, let Q $= \{q_0, \ldots, q_c\}$ be the state set. Let $|y|$ denote the length (in binary) of y. By assumption, there exists constants k, c, such that M computes $f(x)$ in number of steps bounded by $O(2^{2^{\cdot^{\cdot^{\cdot^{2^{x^k}}}}}})$, where the height of the tower of 2's is c. Let W denote the input number. All variables, here, are *integer* typed. Let y be any integer and let HYPEREXP(y) return a *list* of length $O(2^{2^{\cdot^{\cdot^{\cdot^{2^{y^k}}}}}})$, for some constants k, c, where the height of the tower of 2's is c. Given x, it is easy to express in C an integer *poly*(x) such that *poly*(x) = x^k. Let Exp(y) = 2^y. Then, it is easy to compute HYPEREXP(y) by composing the function Exp() c times:

$$\mathrm{E}(y) \equiv \text{TC-INT}(0 \to 0, \lambda(i, r) succ_1(r))(y)$$

$$\mathrm{Exp}(y) \equiv succ(\mathrm{E}(y))$$

$$\mathrm{HYPEREXP}(y) \equiv \mathrm{EXP}(\mathrm{EXP} \ldots (\mathrm{EXP}(poly(y))) \ldots)$$

Next we simulate M by using variables: L to represent the contents of the tape to the left of the head, R to represent the contents to the right, h to represent the content under the tape head, Q the state of the machine. We can now express M's computation as:

```
TC-INT($0 \rightarrow (0, W\text{mod } 2, W\text{div } 2, q_0)$,
    $\lambda(a, L, h, R, Q)$ if $\delta_M(Q, h, q', \ , \text{L})$
                        then $(L\text{div } 2, L\text{mod } 2, 2R + h, q')$
                        $\vdots$
                        else if $\delta_M(Q, h, q', \ , \text{R})$
                        then $(2L + h, R\text{mod } 2, R\text{div } 2, q')$
                        $\vdots$
                        else if $\delta_M(Q, h, q', \sigma, \text{S})$
                        then $(L, \sigma, R, q')$
                        $\vdots$
    ) (HYPEREXP($W$))
```

The IF-THEN-ELSE's are a finite constant sized block of statements depending on δ_M.

In the other direction, since FHYPEREXP is closed under composition and it contains the initial functions of UTC($int, pred, succ_1, succ_0$), it suffices to show that it is closed with respect to the TC-INT recursion. Let k, c be any constants. Let $T_{k,c}(x) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{x^k}}}}}}$, where the height of the tower of 2's is c. Let g, h be functions in FHYPEREXP. Then, there exist functions, T_g, T_h which bound the values of g, h respectively. Analogous to Lemmae 4.2, 4.3, we can show by induction on the depth of derivation of f that the value of any function

$$f(y, \bar{x}; \bar{a}) \equiv \text{TC-INT}(0 \rightarrow g(\bar{x}; \bar{a}), \lambda(i, r)h(i, \bar{x}; r, \bar{a})(y))$$

computed in this language is bounded by

$$f(y, \bar{x}; \bar{a}) \leq yT_h(y, \bar{x}) + T_g(\bar{x}) + \max_i(a_i)$$

If

$$f(\bar{x}, \bar{y}) \equiv h(g(\bar{x}), \bar{y})$$

then

$$f(y, \bar{x}; \bar{a}) \leq T_h(T_g(\bar{x}), \bar{y})$$

The proof is then completed by noting that the inputs, say, $\bar{x}$, are normal i.e. they can be recursed upon and hence by induction, $y, \bar{a} \leq \text{HYPEREXP}(poly(\bar{x}))$. Thus, $f(\bar{x})$ can be computed by a Turing machine in time hyperexponential in the input length. ∎

5. Conclusion

The complexity of UTC(*int*) and UTC(*list*) are open. Multiplication can be expressed in both languages. UTC(*list*) contains REG, the class of regular languages. It seems to be incomparable to FO. It would be nice to show, for example, that equality of two input lists cannot be done in UTC(*list*). UTC(*list*, =) is another puzzling class. It contains FO and REG but seems to very weak otherwise.

For any *integer* i, the operator FOLD is defined as follows:

$$\begin{aligned} \mathrm{FOLD}(f,g)(i) &\equiv (zero \rightarrow g(), \lambda(r)\ f(r))(i) \\ &\equiv f((f(\ldots, f(g())\ldots))) \end{aligned}$$

The variable r is called an accumulative variable. Note that FOLD differs from TC-INT in that it cannot recurse on any predecessors of the argument i. Let FOLD(*int*) denote the class of functions containing the initial functions: zero, projection, successor, conditional, and closed under FOLD operation and composition. Let UFOLD(*int*) denote the uniform version of this language: the accumulative variable r cannot be recursed upon. Further, in this algebra, = and *pred* are not available as initial functions (It follows from our results that UFOLD(*int*, =) or, UFOLD(*int*, *pred*) equal FLS.). The complexity of FOLD and UFOLD(*int*) are open. They contain the class of regular languages. However, multiplication (which is not regular) can be expressed in it, and UFOLD is closed with respect to addition and multiplication.

6. Acknowledgements

References

[BC92] S. Bellantoni and S. Cook, "A New Recursion-Theoretic Characterization of Polynomial Time", preliminary version appeared in *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.

[B92] S. Bellantoni, "Predicative Recursion and Computational Complexity", *Technical Report* 264/92, University of Toronto, September 1992.

[Co64] A. Cobham, The Intrinsic Computational Difficulty of Functions. *Proceedings of the 1964 Congress for Logic, Philosophy and Methodology of Science*, North Holland, 24-30.

[FSS92] L. Fegaras, T. Sheard, D. Stemple, "Uniform Traversal Combinators: Definition, Use and Properties", *Proceedings of* 11*th Conference on Automated Deduction (CADE-11)*, Saratoga Springs, New York, 1992.

[FS93] L. Fegaras, T. Sheard, "A Fold for All Seasons", Preprint, Oregon Graduate Institute, 1993.

[Gu83] Y. Gurevich, "Algebras of Feasible Functions", *Proceedings of 24th IEEE Symposium on Foundations of Computer Science*, October 1983, 210-214.

[Imm82] N. Immerman, "Relational Queries Computable in Polynomial time", *Proceedings of the 14th ACM STOC*, May 1982, 147-152. Revised version appeared in *Information and Control 68*, 1986, 147-152.

[IPS91] Neil Immerman, Sushant Patnaik, and David Stemple, "The Expressiveness of a Family of Finite Set Languages," *Tenth ACM Symposium on Principles of Database Systems*, 1991, 37-52.

[L92] D. Leivant, "A Foundational Delineation of Poly-time", *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, (1991).

[R] H. E. Rose, "Subrecursion: Functions and Hierarchies", *Oxford Logic Guides 9*, Clarendon Press, Oxford, 1984.

[Va82] M.Y. Vardi, The Complexity of Relational Query Languages. *Proceedings of 14th ACM STOC*, May 1982, 137-146.