TECHNICAL REPORT NO. 621

Polynomial-Time Query Languages for Untyped Lists

by

Edward L. Robertson

Lawrence V. Saxton

Dirk Van Gucht

December 2005

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Polynomial-Time Query Languages for Untyped Lists

Edward L. Robertson[1]
Computer Science Department
Indiana University
robertson@cs.indiana.edu

Lawrence V. Saxton[2]
Computer Science Department
University of Regina
saxton@cs.uregina.ca

Dirk Van Gucht[1]
Computer Science Department
Indiana University
vgucht@cs.indiana.edu

Abstract.

We present a language for querying list-based complex objects. These objects are constructible with untyped nodes and hence permit arbitrary-depth sublists. The language is shown to express precisely the polynomial-time generic functions. The language controls complexity by carefully restricting the replication of values and limiting the form and nesting of recursion.

1. Introduction

The main purpose of a database language is to query and manipulate *collections* of objects. The nature of the collections (sets, multisets, lists etc) that the language manipulates and the way it deals with the individuals in the collection determine the capability of a language. Balancing these two factors to achieve the right mix of expressiveness and resource control has motivated the design of a wide variety of database languages. In this paper we strike such a balance, presenting a language which computes precisely the polynomial-time queries on untyped lists.

Traditionally, the focus of query languages has been on the *set*-type collection type. The relational model was one of the earliest proposals for reasoning about data represented as sets of tuples. Relational algebra and calculus were proposed as simple and elegant means of querying data represented as relations. These languages are, however, extremely limited in expressive power. In order to deal with more interesting classes of queries several languages have been proposed extending relational algebra/calculus. Of particular interest are languages that capture the classes of queries corresponding to *feasible* (i.e., PTIME) computation over flat relations ([8,14]) and nested relations ([13,6]).

Although set-based models are well suited for reasoning about unordered and duplicate-free collections, many applications require support for other types of collections, such as bags and lists. Lists, in particular, have gained significant importance in recent years due to the emergence of XML databases. While the canonical model of XML is a tree, the fact that nodes are of unbounded degree (as in "a book has arbitrarily many chapters") and that descendents of a node are ordered means that the natural implementation of an XML tree is in fact via lists. Specifically, an XML database is an *ordered sequence* (list) of *items*. An item is either an atomic value or a *node*, which, most typically is a named list [4]. Hence, query languages for XML databases manipulate lists. The most prominent and practical among these languages is XQuery [4].

As mentioned earlier, a desirable quality of a query language is that it strikes a good balance between expressiveness and computational feasibility. For set-based query languages this balance is set at the PTIME queries. For list-based querying XQuery does not strike this balance since it permits the specification of arbitrary recursive functions ([4] Section 3.2).

Recently, Neven and Schwentick introduced alternative approaches to query XML databases [12]. In their framework, XML databases are modeled as *trees*. To specify queries over trees, they consider a variety of tools: logic, formal languages, and automata theory. From logic, they considered *monadic second-order logic* (MSO), which is first-order logic augmented with quantification over sets. (Typically, such sets are collections of tree nodes). From formal languages, they considered *regular tree languages*, of which regular languages are a special case (strings are trees of height 1). From automata theory, they considered *tree automata*, which are generalizations of finite automata. Neven and Schwentick studied the relationships between these tools as formal query languages for XML databases. Most recently, Neven [11] considered *tree walking automata*, wherein during a state transition, the automaton can move left, right, up, or down in a

[1] Supported by NSF grant IIS-0082407
[2] Supported by NSERC Discovery Grant

tree. Of most interest for this paper is his result that when one suitably augments these automata with a finite number of relation registers and the ability to introduce tree-node identifiers (id's) one obtains query languages that capture precisely a variety of query complexity classes (PTIME, PSPACE, AND EXPTIME).

In this paper we consider a different approach emanating from standard programming language theory. We consider that XML trees are represented using (untyped) lists and we formulate queries in a restricted functional programming language, `RLL`. `RLL` captures precisely the PTIME queries over untyped lists. The restrictions that are built into `RLL`control complexity by (1) carefully restricting the replication of values and (2) limiting the form and nesting of recursion. We think that introducing this approach to the formal study of querying XML databases adds a useful and insightful alternative to the approaches of Neven and Schwentick.

Researchers have previously considered PTIME computation over lists. In [3] we presented a language that captures precisely the PTIME queries for fixed-depth lists. That work is of limited interest for XML because XML databases are frequently not fixed-depth.

Our approach is most closely related to the work of Bellantoni and Cook [1] and Leivant [9]. These authors introduced restricted functional programming languages for PTIME computations on binary strings. Complexity in these language is controlled by a limited form of *recursion on notation*. Because the base functions on binary strings only take one argument as input, these authors did not have to consider the problems that are caused in controlling complexity through the replication (doubling) of values in recursive computations. Caseiro was the first to deal with this doubling-recursion problem when considering the design of a PTIME language for *binary* trees and *flat* lists [2]. More recently, Hoffman [7] used *linear logic* as a formal tool to deal with these same issues. Even though our work is strongly related to this work, it differs in a variety of aspects: (1) we deal with arbitrary untyped lists rather than just binary trees or flat lists, (2) we characterize the *generic* PTIME list functions (generic functions can not interpret atomic values), and (3) our syntactic restrictions to deal with the doubling-recursion problem are simpler and more *from-first-principles* than either Caseiro's or Hoffman's.

Section 2 contains some general definitions and notations about lists and list functions. In section 3, we introduce the Primitive Recursive List language (`PRL`). `PRL` is a powerful list manipulation language: primitive recursive list functions can be defined in it. In section 4, we restrict `PRL` to the language `RLL`, wherein only PTIME list functions can be defined. In section 5 we illustrate `RLL` with some interesting programs. Finally, in sections 6 and 7 sketch an `RLL` simulation for generic Turing machine computations on lists.

2. Preliminaries

The set of untyped lists used in this paper, denoted $\mathcal{L}$ and called the *hereditarily finite lists*, is constructed from a countably infinite set $\mathcal{U}$ of *atoms* and the empty list ϵ. $\mathcal{L}$ is defined by

$$\begin{aligned} \mathcal{L}_0 &= \mathcal{U} \cup \{\epsilon\} \\ \mathcal{L}_{i+1} &= \mathcal{L}_i \cup \{⟦\ell_1, \ldots, \ell_n⟧ : \text{ each } \ell_j \in \mathcal{L}_i \text{ and } n \in \mathbb{N}\} \\ \mathcal{L} &= \textstyle\bigcup_{i\in\mathbb{N}} \mathcal{L}_i \end{aligned}$$

An equivalent characterization for $\mathcal{L}$ is the smallest set that contains $\mathcal{U}$ and ϵ and that is closed under $⟦\cdot⟧$ (for $\ell \in \mathcal{L}$, $⟦\ell⟧$ is the list containing exactly ℓ) and `append`(formally defined below). It is worthwhile to include ϵ in $\mathcal{L}_0$ for the following definition of "depth".

2.1. *Definition*: A list $x \in \mathcal{L}_i - \mathcal{L}_{i-1}$, for $i \geq 1$, is said to be of *depth i*. Lists in $\mathcal{L}_0$ (including ϵ, the empty list) have depth 0. This certainly captures the notion of *list depth* and justifies our claim of working with arbitrary depth lists.

The implementation model for $\mathcal{L}$ uses Lisp's *constructor* or `cons`-cell. In concept, a `cons`-cell is an ordered pair of lists; in implementation, it is a "`struct`" with two pointers. In turn, `cons`-cells are used to encode binary trees which in turn encode lists. A list is represented as a right-linear binary tree; elements of a list, including sublists, down left links. As with Lisp, the binary tree that encodes a list is always terminated, on the right, with an empty component (in implementation terms, a null pointer). Figure 1 shows a list (from $\mathcal{L}$), the representation of that list using `cons`, and a representation as conceptualized in XML. In the following, we completely ignore the distinction between a member of $\mathcal{L}$ and its unique representation in this implementation.

$\llbracket \llbracket$ "a", "sublist" $\rrbracket$, "note", "null", "pointer" $\rrbracket$

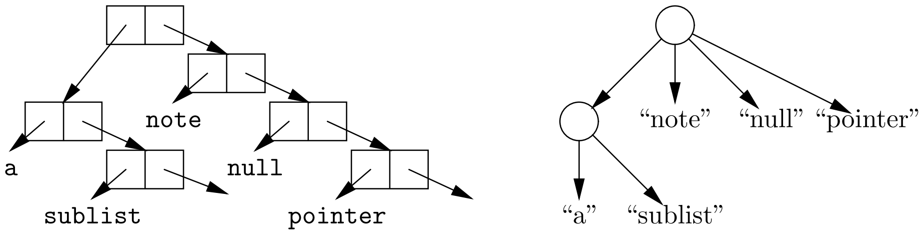

Figure 1. A list, its `cons` representation, and its XML representation.

2.2. *Definition*: The *size* and *length* of a list x, denoted $|x|$ and $len(x)$ respectively, are defined in the obvious way:

$\lvert x \rvert$	$len(x)$	
1	0	when x is ϵ or an atom
$\lvert y_1 \rvert + \lvert y_2 \rvert$	$1 + len(y_2)$	when x is $\mathtt{cons}(y_1, y_2)$

Note that $|\cdot|$ only counts the leaves of a structure (considering ϵ as a leaf), making analysis such as Proposition 4.11 somewhat simpler, since $|\llbracket x,\, y \rrbracket| \;=\; |x| + |y|$. The other natural definition for size is only a factor of 2 larger, since the number of leaves is one greater than the number of internal nodes (as `cons` is binary). Also note that the size of a flat list (in $\mathcal{L}_1 - \mathcal{L}_0$) is one more than its length; for example, $|\llbracket\, \epsilon,\, \epsilon \rrbracket| \;=\; |\mathtt{cons}(\epsilon, \mathtt{cons}(\epsilon, \epsilon))| \;=\; 3$.

2.3. *Definition*: A function $f : \mathcal{L}^n \to \mathcal{L}$ is *generic* iff, for every ζ that is a permutation of $\mathcal{U}$ extended to $\mathcal{L}$ and every $\ell_1, \ldots, \ell_n \in \mathcal{L}$,

$$f(\zeta(\ell_1), \ldots, \zeta(\ell_n)) = \zeta(f(\ell_1, \ldots, \ell_n)).$$

In this paper, we will mainly be concerned with generic list functions that are computable in polynomial time.

Functions are allowed to be partial.

3. The Primitive Recursive List Language

The syntax and semantics of the language `PRL` (Primitive Recursive List) are generally Lisp-like[3], except where expressions are explicitly constrained. There are a few minor syntactic differences from Lisp, such as the use of commas to separate list elements.

The Core of `PRL`

The syntax of the language involves only expressions, function definitions, and the let. The following table gives the details.

<u>expressions</u>		
name	variable	binds to a member of $\mathcal{L}$
ϵ	the empty or "null" list	also written "$\llbracket\,\rrbracket$"
function(*arguments*)	function application	
<u>function definitions</u>		
name	base functions	discussed below
$\lambda\, v_1, \cdots, v_n :\; expression$	λ-expression	
$\lambda\, v_1, \cdots, v_n : [let;]^*\; expression$	λ-expression	
$\mathcal{RS}[function,\; function]$	recursion schema	defined in 3.5
<u>let</u>		
`let` $v_1,\; v_2 \leftarrow v_3$		semantics given in §4.**B**

The language is entirely functional. That is, λ-expressions and let bind new variables, so assignments occur without side effects. Obviously, these variables are the only ones allowed in expression that is the body of the function. In the future, we call this expression the "body expression" of the function.

[3] Knowledge of Lisp is not essential for following this paper.

As with any good programming environment, we assume there is a macro expander that allows us to name function definitions, use meaningful abbreviations, and otherwise apply "syntactic sugar" to our programs. For example, given "$f \stackrel{\text{def}}{=} \lambda x : e$", the macro processor binds f and replaces occurrences of f with $\lambda x : e$.

Base Functions of `PRL`

`PRL` has a a limited set of base functions which provide the computational capability of the language, while allowing control over complexity. The essential syntax and semantics of these functions are defined in the following table, then more subtle details are discussed. Finally, having defined these functions, we turn to the details of the recursion schema $\mathcal{RS}$.

We use "$\perp$" to indicate an explicit error condition. Overriding any other semantics, $\perp$ is returned by any function with $\perp$ for any argument. In addition, `t` is an abbreviation for `cons`(ϵ,ϵ), intuitively "true".

<u>structural base functions</u>			
`cons`	$\stackrel{\text{def}}{=}$	$\lambda\, x, y$: a `cons`-cell, as above	
`append`	$\stackrel{\text{def}}{=}$	$\lambda\, x, y$: y appended to x	
`first`	$\stackrel{\text{def}}{=}$	$\lambda\, e : e_1$ if $e =$ `cons`(e_1, e_2), $\perp$ otherwise	
`rest`	$\stackrel{\text{def}}{=}$	$\lambda\, e : e_2$ if $e =$ `cons`(e_1, e_2), $\perp$ otherwise	
<u>control base functions</u>			
`atom?`	$\stackrel{\text{def}}{=}$	$\lambda\, e : \epsilon$ if $e = \epsilon$ or $e =$ `cons`(x, y), `t` otherwise	
`equal?`	$\stackrel{\text{def}}{=}$	$\lambda\, x, y$: `t` if $x = y$, ϵ otherwise	
`null?`	$\stackrel{\text{def}}{=}$	$\lambda\, e$: `t` if $e = \epsilon$, ϵ otherwise	
`if-then-else`	$\stackrel{\text{def}}{=}$	$\lambda\, c, r_{\mathrm{T}}, r_{\mathrm{F}} : r_{\mathrm{T}}$ if $c \neq \epsilon, r_{\mathrm{F}}$ otherwise	commonly written on three lines
<u>other base functions</u>			
`I`	$\stackrel{\text{def}}{=}$	$\lambda\, x : x$	identity
`min`	$\stackrel{\text{def}}{=}$	$\lambda\, x, y : x$ if $\lvert x\rvert \leq \lvert y\rvert, y$ otherwise	see 3.1 below
`flatten`	$\stackrel{\text{def}}{=}$	$\lambda\, x$: defined in 3.2 below	

While `append` can easily be implemented using `cons` and $\mathcal{RS}$ recursion, providing `append` as a base function greatly simplifies presenting complexity results.

3.1. *Discussion*: The function `min` is not required for the theoretical results of this paper but it is a concession (albeit small) toward programmability. It provides a way to "reuse" storage in a controlled way. The first argument of `min` is typically underlined, as in `min`($\underline{x}$,y). This peculiar notational mnemonic, which is intended to remind us of the way the first argument is used is further explained in 4.8 below. The important facts about `min`($\underline{x}$,y) are that (1) $\lvert$`min`$(\underline{x}, y)\rvert \leq \lvert y\rvert$ and (2) $\lvert x\rvert \leq \lvert y\rvert$ implies `min`$(\underline{x}, y) = x$. The analogous function `max` (replace $\leq$ by $\geq$) does not need to be a primitive because it does not play a role in the complexity control.

3.2. *Definition*: The function `flatten` takes a list structure of arbitrary depth and returns a list of depth two which encodes the structure of the original list. That is, the result of `flatten`(x) is a linear, parenthesized representation of x with special list terms replacing the parentheses. We use the symbols $\triangleleft$ and $\triangleright$ as abbreviations for the lists that encode left and right parentheses respectively. Because we have no reserved characters, we may not use atoms for $\triangleleft$ and $\triangleright$. The natural choice is to use $\llbracket\epsilon\rrbracket$ and $\llbracket\epsilon,\epsilon\rrbracket$ for $\triangleleft$ and $\triangleright$ respectively, thus resulting in a list of depth two.

3.3. *Example*: `flatten`($\llbracket a, \llbracket \llbracket b \rrbracket, c \rrbracket \rrbracket$) is $\llbracket a, \triangleleft, \triangleleft, b, \triangleright, c, \triangleright, \rrbracket$

3.4. *Proposition*: For all x, $\lvert$`flatten`$(x)\rvert \leq 3\lvert x\rvert$.

The `PRL` Recursion Scheme

3.5. *Definition*: The (primitive) recursion scheme, denoted $\mathcal{RS}$ or $\mathcal{RS}[g, h]$, defines a function f from g and h as equivalent to the macro λ-expression

$f \stackrel{\text{def}}{=} \lambda\, y, x :$ `if-then-else( null?`(x),

$$g(y),$$
$$h(\texttt{rest}(x), y, \texttt{first}(x), f(y, \texttt{rest}(x))))$$

We will sometimes write such macro expressions as syntactic sugar for $\mathcal{RS}$ but always understand that the underlying syntax and semantics is $\mathcal{RS}$. Note that we place the recursion variable last, following a convention that places λ-variables according to the way they are used in the bound expression (the distinction between "returnable" and "examinable" variables is discussed in section 4.**B**). The scheme also permits a vector $y_1, \ldots y_k$ in place of y or omission of y entirely.

3.6. *Definition*: The language Primitive Recursive List, denoted `PRL`, is defined by the above core constructors using the above base functions.

3.7. *Proposition*: `PRL` only defines generic list functions (Definition 2.3).

This follows from the fact that the base functions are generic and a simple inductive proof on the structure of the language.

Language Extensions of `PRL`

A few natural extensions of the language, abbreviations which are provided by the macro preprocessor, are treated as if they are part of the language itself. We now discuss those extensions, again giving the essential detail in a concise table followed by discussion. "`t`" is an abbreviation for $\texttt{cons}(\epsilon,\epsilon)$, pronounced "true".

<u>defined functions</u>

$$\texttt{not} \stackrel{\text{def}}{=} \lambda x : \texttt{equal?}(x, \epsilon)$$
$$\texttt{nil}_i \stackrel{\text{def}}{=} \lambda\, v_1 \ldots v_i : \epsilon$$
$$\pi_i^n \stackrel{\text{def}}{=} \lambda v_1, \cdots, v_n : v_i$$

<u>defined functions (infix)</u>

$$\& \stackrel{\text{def}}{=} \lambda x, y : \texttt{if-then-else}(x, \texttt{if-then-else}(y, \texttt{t}, \epsilon), \epsilon)$$
$$\vee \stackrel{\text{def}}{=} \lambda x, y : \texttt{if-then-else}(x, \texttt{t}, \texttt{if-then-else}(y, \texttt{t}, \epsilon))$$

<u>macros requiring recursive expansion</u>

$$\texttt{cond}((c_1, e_1), \cdots, (c_n, e_n)) \stackrel{\text{def}}{=} \begin{cases} \texttt{if-then-else}(c_1, e_1, \texttt{cond}((c_2, e_2), \cdots, (c_n, e_n)), \text{ when } n \geq 1 & \\ \texttt{first}(\epsilon), \text{ when } n = 1 & \text{returns } \bot \end{cases}$$

$$[\![\, \ell_1, \ell_2, \cdots, \ell_k \,]\!] \stackrel{\text{def}}{=} \begin{cases} \texttt{cons}(\ell_1, [\![\ell_2, \cdots, \ell_k]\!]\,), \text{ for } k > 1 \\ \texttt{cons}(\ell_1, \epsilon), \text{ for } k = 1 \\ \texttt{cons}(\epsilon, \epsilon), \text{ for } k = 0 \end{cases}$$

$$\texttt{append}(\ell_1, \ell_2, \cdots, \ell_k) \stackrel{\text{def}}{=} \begin{cases} \texttt{append}(\ell_1, \texttt{append}(\ell_2, \ldots, \ell_k)\,), \text{ for } k > 1 & \\ \ell_1, \text{ for } k = 1 & \text{overload append} \end{cases}$$

<u>composition</u>

$$f \circ g \stackrel{\text{def}}{=} \lambda v_1, \cdots, v_n : f(g(v_1, \cdots, v_n))\text{when } g \text{ is } \lambda v_1, \cdots, v_n : e_1 \text{ and } f \text{ is } \lambda x : e_2$$

4. Complexity Restrictions on `PRL`: The `RLL` Language

In this section we introduce a sublanguage of `PRL`, called `RLL` for `R`eplication `L`imited `L`ist, wherein only polynomial-time list functions can be specified. The difficulty in designing such a language is to introduce formalisms to control the growth of intermediate data structures and the recursive iteration through such structures.

The primary concern is to avoid the introduction of a doubling operation in the context of recursion, since iteration of this apparently innocuous operation can cause exponential blow-up in data structures, clearly an undesirable aspect of a database language that allows it.

Doubling can occur either as the result of the explicit re-use of a variable, as in

$$\texttt{dup} \stackrel{\text{def}}{=} \lambda x : [\![x,\, x]\!]$$

or through replication of the structure, as in

$$\texttt{double} \stackrel{\text{def}}{=} \lambda x : \texttt{if-then-else}(\; \texttt{null?}(x),$$

ϵ,
$\texttt{cons}(\epsilon,\ \texttt{cons}(\epsilon,\ \texttt{double}(\texttt{rest}(x)))))$.

Given a list of n items, `double` returns a list of $2n$ items. When iterated in a recursion, `dup` or `double` can therefore create an exponentially larger object in the size of the input object.

There are three separate factors which must be controlled to achieve the desired control on data structure size. These are (a) the use of the constructor operation `cons` and `append`, (b) the explicit reuse of variables, and (c) the repetition through recursion. Although it is in fact the interaction of these three factors which is dangerous, they can be controlled relatively independently, as follows:

(A) `cons` and `append`

The controls on `cons` or `append` apply to their use in recursion, as the above example illustrated.

However, a computation involving `cons` and `append` outside of the recursion is always polynomial, and a recursion without `cons` or `append`, which can be used to examine and decompose structures, will never increase the size of the structure. To this end, we define `cons`-*free recursion* as any expression not involving `cons` or `append` (including use of the recursion schema $\mathcal{RS}$). Obviously `cons`-free recursion cannot increase the size of its arguments.

4.1. *Example*: The (Boolean) function `in_list?`, such that $\texttt{in\_list?}(e,\ (\ell_1, \ldots, \ell_k))$ is `t` if e is one of the ℓ_i and ϵ otherwise, is `cons`-free:

$$\texttt{in\_list?} \stackrel{\text{def}}{=} \lambda e, \ell :\ \texttt{if-then-else}(\,\texttt{null?}(\ell),\ \epsilon,\ \texttt{if-then-else}(\texttt{equal?}(e, \texttt{first}(\ell)),\ \texttt{t},\ \texttt{in\_list?}(e, \texttt{rest}(\ell))))$$

4.2. *Definition*: The *computation time* to evaluate an expression $e = f(e_1, \ldots, e_k)$, denoted $\mathcal{T}(e)$, is the number of base function calls in the evaluation of e. Thus, $\mathcal{T}(e)$ is the sum $\mathcal{T}(e_1) + \ldots + \mathcal{T}(e_k)$ plus the calls used in the evaluation of f on the values of $e_1, \ldots, e_k$. If f is $\texttt{if-then-else}(c, e_1, e_2)$, then $\mathcal{T}(f)$ is $1+\mathcal{T}(c)+\mathcal{T}(e_i)$, where i is 1 if c is true or i is 2 if c is false respectively. If f is defined by a recursion schema, $\mathcal{T}(e)$ is defined by expanding the recursion in the natural way [4].

4.3. *Proposition*: If f is defined without $\mathcal{RS}$, or f is `cons`-free, then there are k_1 and k_2 such that

$$\mathcal{T}(f(x_1, \cdots, x_n)) \leq k_1 \times \prod |x_i|^{k_2}.$$

(B) restrictions on reuse of variables

The problem of doubling discussed at the beginning of this section is avoided if we explicitly require that a variable cannot be reused. But because variables typically contain complex objects, this requirement is too restrictive in two ways. Each excessive restriction requires a carefully controlled relaxation. The first excessive restriction does not allow use of the distinct *pieces* of the object in different places.

4.4. *Example*: Consider a function that performs a kind of list rotation, for instance transforming

$$\begin{array}{c} [\![\ \downarrow\ , \mathbf{q}, r, s, \ldots]\!] \\ [\![a, b \ldots]\!] \end{array} \quad \text{into} \quad \begin{array}{c} [\![\ \downarrow\ , r, s, \ldots]\!] \\ [\![\mathbf{q}, a, b, \ldots]\!] \end{array}$$

The downward arrow indicates a sublist. This function is easy to express, as

$$\texttt{rotl} \stackrel{\text{def}}{=} \lambda w :\ \texttt{cons}(\texttt{cons}(\texttt{first}(\texttt{rest}(w)), \texttt{first}(w)), \texttt{rest}(\texttt{rest}(w))),$$

but the variable w (or some other variable if there are nested λ expressions) must be repeated. It obviously does not change the size of its argument because $\texttt{first}(w)$, $\texttt{first}(\texttt{rest}(w))$, and $\texttt{rest}(\texttt{rest}(w))$ are indeed distinct parts of w. Generalizing this solution involves some bookkeeping in order to have explicit control over the use of variables.

To achieve this bookkeeping, we introduce a slight variation in function definition. The traditional λ-expression is extended to

$$\lambda\langle\text{variables}\rangle :\ [\langle\text{let}\rangle;]^*\ \langle\textit{expression}\rangle.$$

[4] The implementation of the base functions on a Turing machine, or other implementation model, requires time at worst a low-order polynomial in the computation's space.

A let (named for Lisp's `let`) has syntax `let` $v_1,\ v_2 \leftarrow v_3$, where the v_i are variables, and semantics which assigns $\texttt{first}(v_3)$ to v_1 and $\texttt{rest}(v_3)$ to v_2. Consistent with the other primitives, a let returns $\perp$ to both v_1 and v_2 if v_3 is not a `cons` expression. Variables appearing on the left-hand-sides of lets, which we refer to as *local* variables, are implicitly declared and must not appear in the parameter list. Variables in the right-hand-sides of lets must have been defined previously in the function definition, either as formal parameters or as local variables. All variable scoping is strictly local.

Using this notation, the function of example 4.4 may be expressed as

$$\begin{array}{ll} \lambda w: & \texttt{let}\ x,\ w_r \leftarrow w; \\ & \texttt{let}\ y,\ z \quad \leftarrow w_r; \\ & \texttt{cons}(\texttt{cons}(y,\ x),\ z) \end{array}$$

Similarly, example 4.1 may be rewritten, adding `let` $f,\ r \quad \leftarrow \ell$ and replacing $\texttt{first}(\ell)$ and $\texttt{rest}(\ell)$, respectively, with f and r.

The second excessive restriction concerns variables which do not appear in the result of an expression (or whose appearance is strictly controlled with `min`). To provide the necessary flexibility, parameters of functions and the variables of expressions are partitioned into *returnable* and *examinable* classes through the following definitions. Returnable variables are ones whose values may be **returned** as or within the results of computations. The intuition for calling the remaining variables "examinable" is that they may be looked at but not used in constructing results. That is, their space may not be used.

The base function `min` is a somewhat peculiar case. Although the value returned by `min` may be e_1 or e_2, the space allowed for the result is at most that of e_2. Thus, for `min`, e_1 is said to be examinable and e_2 returnable.

The following two definitions are mutually recursive, respectively applying to expressions and function definitions. Since this recursion merely follows the way in which items of each sort appear inside the other, the definition is well-founded.

4.5. *Definition*: The set of *returnable variables* of an expression e, denoted $\mathrm{ret}(e)$, is defined by the following table. The table also gives equivalent characterizations for returnable variables for some of the "macro" constructions.

form of e	returnable variables	
the single variable x	$\{x\}$	
$\texttt{first}(e_1)$, $\texttt{flatten}(e_1)$, or $\texttt{rest}(e_1)$	$\mathrm{ret}(e_1)$	
$\texttt{min}(\underline{e_1},e_2)$	$\mathrm{ret}(e_2)$	
$\texttt{cons}(e_1, e_2)$	$\mathrm{ret}(e_1) \cup \mathrm{ret}(e_2)$	
`if-then-else` $(c_1,\ r_1,\ r_2)$	$\mathrm{ret}(r_1) \cup \mathrm{ret}(r_2)$	
$f(e_1, \ldots, e_n)$ and R is the set of indices of returnable parameters of f	$\bigcup_{i \in R} \mathrm{ret}(e_i)$	application of function f defined by λ
otherwise	$\emptyset$	
derivable forms		
$\texttt{not}(e)$	$\mathrm{ret}(e)$	
$\texttt{nil}_i(e_1, \cdots, e_i)$	$\emptyset$	
$\pi_i^n(e_1, \cdots, e_n)$	$\mathrm{ret}(e_i)$	
$\texttt{cond}(\ (c_1,\ r_1),\ \cdots\ (c_n,\ r_n)\ )$	$\bigcup_{1 \le i \le n} \mathrm{ret}(r_i)$	

4.6. *Definition*: The *returnable parameters* of a function f defined by a λ expression are those formal parameters which are returnable variables of the expression or which appear in the right-hand-sides of lets.

Recalling that $\mathcal{RS}$ is a syntactic constructor, the definition of recursion for $f = \mathcal{RS}[g,\ h]$ obviously depends upon the nature of g and h. If the second parameter of h is returnable, it is easy to do doubling.

Thus we consider the case where the first two parameters of h are only examined. This restriction appears in definition 4.12 below, which provides the major connection between recursion syntax and complexity. In this case, if $f = \mathcal{RS}[g, h]$, then $ret(f(x)) = ret(g(y)) \cup ret(x)$. This can be derived as $ret(f(x, y)) = ret(g(y)) \cup ret(\texttt{first}(x)) \cup ret(\texttt{first}(\texttt{rest}(x))) \cup ret(\texttt{first}(\texttt{rest}(\texttt{rest}(x)))) \cup \cdots = ret(g(y)) \cup ret(x)$.

4.7. *Notation*: If a variable or parameter appears in an expression or function, respectively, but is not returnable, then it is *examinable*.

4.8. *Notation*: As an aid in distinguishing returnable and examinable parameters, we adopt the convention of writing all examinable parameters first in a function invocation and indicate those parameters $\underline{\text{thusly}}$. That is, $f(\underline{w_1, \ldots, w_m}, x_1, \ldots, x_n)$ indicates that f has m examinable parameters and n returnable ones. This convention is sometimes extended to λ expressions and to indicate examinable subexpressions in function definitions. Note that this is not part of the formalism but merely a bookkeeping aid. Returning to our previous examples, we may write invocation of `in_list?`, example 4.1, and `rotl`, 4.4, as `in_list?`$(\underline{e, \ell})$ and `rotl`(ℓ) respectively. This raises an interesting point about `in_list?`, namely that it returns neither argument but only ϵ or `t`.

The following two definitions are mutually recursive. Since this recursion merely follows the way in which items of each sort appear inside the other, the definition is well-founded.

4.9. *Definition*: An expression e without $\mathcal{RS}$ is *safe*

form of e	when e is safe
the single variable x or ϵ	always
`first`(e_1) or `rest`(e_1)	if e_1 is safe
`cons`(e_1, e_2) or `append`(e_1, e_2)	if both e_1 and e_2 are safe and $\mathrm{ret}(e_1) \cap \mathrm{ret}(e_2) = \emptyset$
`min`$(\underline{e_1}, e_2)$	if e_2 is safe
`cond`$(\ (c_1, r_1),\ \cdots\ (c_n, r_n)\ )$	if each r_i is safe
`flatten`(ℓ)	never
$f(e_1, \ldots, e_n)$ and R is the set of all returnable parameters of f	if f is safe and each e_i is safe, for all $i \in R$, and $\mathrm{ret}(e_i) \cap \mathrm{ret}(e_j) = \emptyset$, for $i, j \in R$ and $i \neq j$

4.10. *Definition*: A defined function f is *safe* if (1) each returnable or local variable appears at most once in either a returnable position in the body expression or the right-hand-side of a let and (2) the expression defining the function is a safe expression. A function defined by `cons`-free recursion is also safe.

As a convenience we will use "level-i expression" to mean expressions which contain functions of at most level-i. Note that these levels are parallel to the nesting levels of the loop language of Meyer and Ritchie [10], most easily after unrolling recursions to loops.

Example 4.4, `rotl`, above illustrates the safe definition of a function. Example 4.1 is more interesting, because we cannot syntactically exclude `in_list?`(ϵ,ϵ) from returning `t`, a dangerous situation. Looking at the semantics (*i.e.* evaluating the expression), we see that `in_list?`(ϵ,ϵ) in fact returns ϵ. Thus, although `in_list?` is not safe in our formalism, its behavior is safe-like.

4.11. *Proposition*: If f is a safe function with $w_1, \cdots, w_m$ examinable and $x_1, \cdots, x_n$ returnable, then there is a k such that

$$|f(\underline{w_1, \cdots, w_m},\ x_1, \cdots, x_n)| \leq \sum |x_i| + k.$$

Proof: The first step is to execute all let statements from f. The definition of $|\cdot|$ shows that this does not change the sums (a term may be replaced by two terms together equal to the first).

The proof proceeds by structural induction based upon the definition of f. We explicitly do the one significant case where f is defined as `cons`$(f_1(\cdots),\ f_2(\cdots))$. Say R is the set of indices of returnable parameters of f and R_1 and R_2 are the subsets of R corresponding to the returnable parameters of f_1 and f_2

respectively. Thus

$$\begin{aligned}
|f(\underline{w_1, \cdots, w_m},\, x_1, \cdots, x_n)| &= |f_1(\cdots)| + |f_2(\cdots)| \\
&\leq \sum_{i \in R_1} |x_i| + k_1 + \sum_{j \in R_2} |x_j| + k_2 \\
&\leq \sum_{i \in R} |x_i| + k
\end{aligned}$$

The last inequality holds because f is safe and thus $R_1 \cap R_2 = \emptyset$. It is not equality because $R_1 \cup R_2$ may be a proper subset of R.

Note that returnability, examinability, and safety are entirely syntactic constructs. The rules that define them can easily be evaluated on the parse tree of an expression.

(C) recursion

The use of mechanisms that monitor the reuse of variables must be accompanied by an equivalent care with nestings of recursion. For example, the `double` function, defined at the beginning of this section, has only safe operations (other than the recursion scheme) but the nested recursion $\mathcal{RS}[\mathtt{nil}_1, \mathtt{double}]$ returns a result which is exponentially larger than the input.

4.12. *Definition*: A function is

- *level*–0 if it is a safe function (including `cons`-free recursion) and expressions occurring at examinable positions of other functions in the body expression are at most level-1;
- *level*–i, for $i \geq 1$, if it is level–(i-1), or is defined by $\mathcal{RS}$ recursion from level–(i-1) functions (where only the last two arguments of the second function are returnable), or it is `flatten`, or it is the composition of level–i functions.

4.13. *Proposition*: If $f = \mathcal{RS}[g, h]$ and g and h are safe then there is an integer k such that

$$|f(y, x)| \leq (k + |x| + |y|) \times len(x)$$

Proof: Let k be the larger of the two constants corresponding to g and h from Proposition 4.11. That Proposition also provides the basis for the induction. For the inductive step, let $x = \mathtt{cons}(x_1, x_2)$.

$$\begin{aligned}
|f(y,x)| &= |h(x_2, y, x_1, f(y, x_2))| && \leq |x_2| + |y| + |x_1| + |f(x_2, y)| + k \\
&\leq |x| + |y| + k + len(x_2) \times (|x_2| + |y| + k) && \leq (len(x_2) + 1) \times (|x| + |y| + k) \\
&= len(x) \times (|x| + |y| + k)
\end{aligned}$$

A structural induction on the definition of level–1 function that makes use of proposition 3.4, proposition 4.11, and 4.13 implies the following:

4.14. *Proposition*: If f is level–1, then there is an integer k such that

$$\mathcal{T}(f(x)) \leq k \times |x|^k.$$

4.15. *Definition*: The language `RLL` is the language of all level–1 `PRL` expressions.

The previous proposition can then be used to prove the following:

4.16. *Theorem*: Every function defined by a `RLL` expression runs in time polynomial in the size of its input.

In actuality we can also prove the reverse of this theorem. In particular, we prove in section 6 that each generic PTIME computable function over lists can be expressed as a `RLL` expression.

The previous theorem and theorem 6.5 can be succinctly stated in our main theorem:

4.17. *Theorem*: PTIME-generic-list functions $\equiv$ `RLL` functions.

5. Some Interesting RLL Examples

5.1. *Constrained Recursion*: A particularly useful form of recursion follows the recursion scheme *constrained recursion*, whose abbreviation as "cons-recursion" is suggestive of the essential role the `cons` operator plays.

5.2. *Definition*: A function f is defined by *constrained recursion* from functions g and h, where h examines its first two arguments, if

$$\begin{aligned} f(y,x) \; = \; & \texttt{if-then-else(}\ \texttt{null?}(x), \\ & \quad g(y), \\ & \quad \texttt{cons(}\ h(\texttt{rest}(x), y, \texttt{first}(x)), f(y, \texttt{rest}(x))\ )\ ) \end{aligned}$$

This schema is denoted $\mathcal{CRS}[g,\, h]$. Again y may be a vector.

It is obvious that $\mathcal{CRS}[g,\, h] = \mathcal{RS}[g,\, h']$ for $h' = \lambda\, x_1, y, x_2, z\ :\ \texttt{cons}(h(x_1, y, x_2), z)$. Moreover, provided that h examiines its first two parameters,

$$ret(h'(x_1, y, x_2, z) = ret(h(x_1, y, x_2)) \cup ret(z) = ret(x_2) \cup ret(z),$$

so h' fits the requirments of definition 4.6 and thus $ret(f(x,y)) = ret(g(y)) \cup ret(x)$. Hence functions definable as i nestings of $\mathcal{CRS}$ as in definition 4.12 above are in fact level–i. $\mathcal{CRS}$ is quite similar to Lisp's `mapcar`, except that the map function gets to examine, but not reproduce, more context.

5.3. *Proposition*: A function $f \stackrel{\text{def}}{=} \mathcal{CRS}[g,h]$ has a level-1 definition if g and h are level-1 and, for all $w,\, y,\, z \in \mathcal{L}$, $|g(y)| \le |y|$ and $|h(w,y,z)| \le |z|$.

Proof: The appropriate level-1 definition is

$$\mathcal{CRS}[\lambda y : \texttt{min}(\underline{g(y)},\, y), \lambda x_f,\, y,\, x_r : \texttt{min}(\underline{h(x_f, y, x_r)}, x_r)]$$

5.4. *Notation*: For an integer n, $\hat{n}$ denotes a list of n empty lists, that is $[\![\, \epsilon \cdots \epsilon \,]\!]$ with ϵ repeated n times. With this representation in mind,

$$\begin{aligned} \texttt{add} &\stackrel{\text{def}}{=} \texttt{append} \\ \texttt{mult} &\stackrel{\text{def}}{=} \mathcal{CRS}[\texttt{nil}_1, \lambda x_1, x_2, x_3, x_4 : \texttt{add}(x_2, x_4)] \\ \texttt{size} &\stackrel{\text{def}}{=} \mathcal{CRS}[\texttt{nil}_1,\, \texttt{nil}_4] \circ \texttt{leaves} \circ \texttt{flatten} \end{aligned}$$

where `leaves` is a simple recursion that discards the encoded left and right parentheses from a flattened list. The function `mult` is unlikely to do anything worthwhile given inputs not of the form $\hat{n}$. However the definition of `size` is indeed an implementation of the size function $|\cdot|$, in that $\texttt{size}(x) = \widehat{|x|}$.

5.5. *Simulation of Cartesian product*: It is often essential to compose several recursions to do the "space allocation" before useful work is done with Proposition 5.3. We illustrate this space allocation with a list version of Cartesian product. Given $\ell = [\![\ell_1, \ldots, \ell_m]\!]$ and $k = [\![k_1, \ldots, k_n]\!]$, $\texttt{pairs}(\ell,\, k)$ produces a list of all $[\![\ell_i,\, k_j]\!]$ pairs. Several utility functions are required.

$$\begin{aligned} \texttt{max} &\stackrel{\text{def}}{=} \lambda x, y : \texttt{if-then-else} x = \texttt{min}(\underline{x}, y) y x \\ \texttt{max\_list} &\stackrel{\text{def}}{=} \lambda \ell : \texttt{let} x,\ y \quad \leftarrow \ell; \mathcal{RS}[\texttt{nil}_1, \lambda d_1, y, d_2, x : \texttt{max}(x,y)](x,y) \\ \texttt{pair\_with} &\stackrel{\text{def}}{=} \mathcal{CRS}[nil_1, \lambda d, m, e, r : \texttt{cons(}\ \texttt{cons}(m,\, e),\, r)]\ (\texttt{max\_list}(\ell),\, k) \end{aligned}$$

So $\texttt{pair\_with}(e,\, k)$ produces $[\![\, [\![e, k_1]\!], \ldots, [\![e, k_n]\!] \,]\!]$.

$$\texttt{alloc} \stackrel{\text{def}}{=} \lambda\, \ell, k : \texttt{pair\_with}(\texttt{max\_list}(\ell), k)$$

Finally,

$$\texttt{pairs} \stackrel{\text{def}}{=} \lambda \ell, k : \mathcal{CRS}[\texttt{nil}_1, \lambda d, y_1, y_2, w : \texttt{min}(\underline{\texttt{pair\_with}(w, y_1)}, y_2)](k, \texttt{alloc}(\ell, k), \ell)$$

Striking the `min` and attendant bookkeeping from the above, we have the natural definition of `pairs` as $\mathcal{CRS}[\texttt{nil}_1, \texttt{pair\_with}]$.

5.6. *Transitive closure*: The transitive closure operation has taken on special significance as characteristic of iterative computations just beyond the power of relational algebra. Thus it is a good vehicle for a larger example. The program for transitive closure is the composition of several steps, interleaving "space allocations" and actual computation.

Transitive closure is defined on a graph represented as a list of pairs. Consider a graph with m vertices $\{v_1, \cdots, v_m\}$ and n edges. Then the graph is represented by a list of pairs

$$\ell_1 \stackrel{\text{def}}{=} \llbracket\, \llbracket v_{s_1},\ v_{d_1} \rrbracket,\ \llbracket v_{s_2},\ v_{d_2} \rrbracket, \cdots \llbracket v_{s_n},\ v_{d_n} \rrbracket\, \rrbracket$$

The first step is to extract from ℓ_1 a list where each source vertex appears once, that is (in some order)

$$\ell_2 \ = \ \llbracket v_1,\ \cdots,\ v_m \rrbracket$$

The next step is to define a function `triples`, much like `pairs` above. Then ℓ_3 is defined as $\texttt{triples}\llbracket \ell_2 \rrbracket$ and ℓ_4 as m copies of ℓ_3. Having defined these structures, the real computation then uses a working list of edges, initially ℓ_1, and iterates through the triples $\llbracket x,\ y,\ z \rrbracket$ in ℓ_4. If $\llbracket x,\ y \rrbracket$ and $\llbracket y,\ z \rrbracket$ are already on the working list but $\llbracket x,\ z \rrbracket$ is not, then $\llbracket x,\ z \rrbracket$ is added. Since each triple appears m times and no path is longer than m, every edge in the transitive closure is eventually added to the working list. Formally,

$$\begin{array}{ll}\texttt{tc\_aux} \stackrel{\text{def}}{=} \lambda\ trpl, edges: & \texttt{let}\ x,\ x_r \ \leftarrow trpl; \\ & \texttt{let}\ y,\ z \quad \leftarrow x_r; \\ & \texttt{if-then-else}(\ \underline{\texttt{in\_list?}((x,y), edges)\ \&\ \texttt{in\_list?}((y,z), edges)}, \\ & \qquad\qquad\qquad\ \ \texttt{set\_add}((x,z), edges), \\ & \qquad\qquad\qquad\ \ edges)\end{array}$$

Then $\texttt{trans\_close}(\ell_1) \stackrel{\text{def}}{=} \mathcal{RS}[\texttt{I}, \texttt{tc\_aux}](\ell_1, \ell_4)$. This construction illustrates the trick that transforms an algorithm with several nested iterations to a program where nesting has been replaced by space allocation and composition.

5.7. `unflatten`: We now turn our attention to unflattening a list. This function is used later as the final step of computing an arbitrary PTIME function, where it is applied after the Turing machine simulation is completed.

To understand `unflatten`, remember that all work in an $\mathcal{RS}$ schema is really done after the return from the recursive step, so we think of a process working from the right of the flattened list. This process uses a "working store" $\mathcal{W}$, which is actually the result returned by `unflatten`. As an example of the contents of and operations on $\mathcal{W}$, consider the list encoding (presented as a string of symbols, without regard for genericity and without commas)

$$\begin{array}{lr}\triangleleft\ a\ \triangleleft\ b\ \triangleright\ \triangleleft\ c \quad\ d\ \triangleright\ e\ \triangleleft\ f\ \triangleright\ \triangleright & \\ \qquad\qquad\quad\ \Uparrow & \longleftarrow \ \text{direction of scan}\end{array}$$

which has been partially scanned up to the point indicated by $\Uparrow$. At this point, there is a partially completed list from depth 1 representing $\triangleleft\, e,\ \triangleleft\, f \triangleright \triangleright$ and a partially completed list from depth 2 representing only $d \triangleright$. In general, $\mathcal{W}$ is a list $\llbracket \ell_i,\ \ell_{i-1}, \ldots,\ \ell_1 \rrbracket$, where each ℓ_j is the partially accumulated result from depth j. At the end of the computation (on a correctly parenthesized string), the result will be $\llbracket\, \ell_1 \rrbracket$; hence, $\texttt{first} \circ \texttt{unflatten}$ will unflatten the flattened list.

Continuing the example, as the scan moves left, it encounters the symbol c. Because c is not list punctuation, it goes on the list from depth 2. Indeed, $\mathcal{W}$ functions as a stack with elements from the deepest level being processed on top. Operations are on the top one or two sublists of $\mathcal{W}$. After c, the scan next encounters "$\triangleleft$". This indicates that the depth 2 sublist is complete and should be placed on the depth 1 list. Next "$\triangleright$" indicates a new sublist from depth 2 should be established, and so on. Thus there are three basic operations:

scanning	name	**operation**	**definition, as** $\lambda\ e,\ \mathcal{W}$
$\triangleleft$	`pop`	ℓ_i moved from front of $\mathcal{W}$ to front of ℓ_{i-1}	$\texttt{F}, \texttt{R} \ \leftarrow\ \mathcal{W}$; $\texttt{FR}, \texttt{RR} \ \leftarrow\ \texttt{R}$; $\texttt{cons}(\texttt{cons}(\texttt{F}, \texttt{FR}), \texttt{RR})$
$\triangleright$	`push`	empty ℓ_{i+1} goes on front of $\mathcal{W}$	$\texttt{cons}(\texttt{min}(\underline{\epsilon}, e), \mathcal{W})$
other	`insert`	current symbol e goes at front of ℓ_i	$\texttt{F}, \texttt{R} \ \leftarrow\ \mathcal{W}$; $\texttt{cons}(\texttt{cons}(e, \texttt{F}), \texttt{R})$

It is easy to see that each of `insert`, `pop`, and `push` is safe. Each has two arguments for conformity and safety, although the e is explicitly replaced by ϵ in `push`.

These operations are then combined into

$$
\begin{array}{l}
\texttt{unflatten} \stackrel{\text{def}}{=} \lambda\, x : \\
\quad \texttt{if-then-else}(\\
\qquad \mathit{null?}(x), \\
\qquad \epsilon, \\
\qquad \texttt{cond}(\ (\texttt{equal?}(\texttt{first}(x), \lhd), \quad \texttt{pop}(\texttt{first}(x), \texttt{unflatten}(\texttt{rest}(x)))) \\
\qquad \qquad\quad (\texttt{equal?}(\texttt{first}(x), \rhd), \quad \texttt{push}(\texttt{first}(x), \texttt{unflatten}(\texttt{rest}(x)))) \\
\qquad \qquad\quad (\texttt{t}, \qquad\qquad\qquad\qquad \texttt{insert}(\texttt{first}(x), \texttt{unflatten}(\texttt{rest}(x)))) \\
\qquad)\ )
\end{array}
$$

This program does no explicit error checking for unbalanced parentheses. However run-time checking will catch most errors and a test at the outside (replaces simple `first`) that the result is exactly (t_0) will catch the others.

6. Simulation of a Time-Bounded Turing Machine

We now must be more specific about Turing machines, their computations, and their representations. Of particular importance is the way in which a list is represented on the Turing machine's tape, namely: the list is linearized with sublists delimited by special symbols and list atoms encoded in unary. This requires an alphabet of four symbols, which correspond to left and right parentheses (for delimiting lists), one (for encoding atoms), and commas (for separating atom encodings). We do not fix particular symbols for this encoding but require that each Turing machine description include a table that specifies these symbols. For readability, we will use "$\lhd$", "$\rhd$", "**1**", and "**,**" for these symbols but understand that each Turing machine has its own vocabulary.

Two mapping functions are required to cast between lists and their string representations: `string2list` and `list2string`. These mappings are of course dependent on the particular encoding used by the Turing machine.[5]

The notation $\mathfrak{M}$ denotes both a Turing machine description (as a seven-tuple, since the description now includes a symbol table) and the string-to-string (partial) function computed by that machine. A generic Turing machine is simply a Turing machine that computes a generic function (as defined toward the end of the Introduction) but genericity on lists requires a bit more care.

6.1. *Definition*: A Turing machine $\mathfrak{M}$ is *list generic* provided that, for every permutation ζ of $\mathcal{U}$ extended to $\mathcal{L}$ and every $\ell \in \mathcal{L}$,

$$\texttt{string2list}(\mathfrak{M}(\texttt{list2string}(\zeta(\ell)))) = \zeta(\texttt{string2list}(\mathfrak{M}(\texttt{list2string}(\ell))))$$

To show that any PTIME computation may be simulated by a level–1 function, we show separate constructions for a clock and for a time-bounded Turing machine simulation. Note that both the machine description $\mathcal{M}$ and the time-bound p must be given, which is to be expected since no function which universally simulates all of PTIME could itself be in PTIME(by diagonalization).

6.2. *Clock*:

We need to produce lists which are polynomial in the size of the input lists, for arbitrary polynomials. Compositions of the arithmetic functions can be used to define functions rep_p, for any polynomial p, such that, for any list x, $|rep_p(x)| = \hat{n}$, where $n > p(|x|)$.

6.3. *Turing machine computation*:

The following discussion simulates the computation of the simplest kind of Turing machine, with one read/write head and hence one tape.

---

[5] An alternative approach is to have the mapping functions declare what symbols they use, so `list2string` would prepend the equivalent of "$\lhd \rhd$ **1 ,**" to the actual list representations.

A Turing machine description, denoted in the following discussion by $\mathcal{M}$, is a list of the state transitions, of start states, and of end states. Each state transition is itself a list of the form $(q,\ \sigma,\ q',\ \sigma',\ \langle move\rangle)$, where q and q' encode machine states, σ and σ' encode tape symbols, and $\langle move\rangle$ is one of $\llbracket\ \epsilon\ \rrbracket$, $\llbracket\ \epsilon,\ \epsilon\ \rrbracket$, or $\llbracket\ \epsilon,\ \epsilon,\ \epsilon\ \rrbracket$ encoding move-left, no-motion, and move-right respectively. In a proper encoding of a machine, we require that $|q| = |q'|$ and $|\sigma| = |\sigma'|$, for all pairs of state and symbol encodings.

The heart of a Turing machine simulation is an iteration of steps, each of which makes one local transformation to the configuration. A configuration includes both tape contents and machine head position/state information. In particular, a configuration is a list of four items:

$$\llbracket\text{state, cell under head, cells left of head, cells right of head}\rrbracket,$$

where the last two items are themselves lists of cells with the cell closest to the head at the front of the list. The full simulation, which supplies the clock controlling the number of iterations, also supplies sufficient blank tape so that the head will not run off that space within *clock* steps.

The general design for the Turing machine simulation is

```
iterate clock steps
        compute next configuration nxtconfig
```

More precisely, given a clock $\hat{n}$, an initial tape, and a machine description $\mathcal{M}$, *simulate* steps through n steps of the Turing machine defined by $\mathcal{M}$.

$$
\begin{array}{l}
simulate \stackrel{\text{def}}{=} \lambda\ \underline{\mathcal{M}},\ config,\ clock :\\
\qquad \texttt{if-then-else}(\ \texttt{null}?(clock),\\
\qquad\qquad config,\\
\qquad\qquad nxtconfig(\underline{\mathcal{M}},\ simulate(\underline{\mathcal{M}},\ config,\ \texttt{rest}(clock))))
\end{array}
$$

We now define *nxtconfig*. The functions *move_right*?, *move_left*?, *next_state*, and *new_cell* are all defined by `cons`-free recursion. To improve readability in the following definition, the result expressions of the `cond` omit the arguments of certain functions and use list notation which does not fully conform to syntactic requirements for safety. After the definition, the first of these result expressions will be expanded into correct syntax.

$$
\begin{array}{l}
nxtconfig \stackrel{\text{def}}{=} \lambda\ \underline{\mathcal{M}},\ config :\\
\qquad \texttt{let}\ state,\ tmp_1 \quad \leftarrow config;\\
\qquad \texttt{let}\ cell\_C,\ tmp_2 \quad \leftarrow tmp_1;\\
\qquad \texttt{let}\ tape\_L,\ tmp_3 \quad \leftarrow tmp_2;\\
\qquad \texttt{let}\ tape\_R,\ \mathcal{X} \quad \leftarrow tmp_3;\\
\qquad \texttt{let}\ cell\_L,\ rest\_L \quad \leftarrow tape\_L;\\
\qquad \texttt{let}\ cell\_R,\ rest\_R \quad \leftarrow tape\_R;\\
\qquad \texttt{cond}(\ (move\_right?(\mathcal{M}, state, cell\_C)),\\
\qquad\qquad\qquad \llbracket next\_state(\ldots),\ cell\_R,\ \texttt{cons}(changed\_cell(\ldots), tape\_L),\ rest\_R\rrbracket\\
\qquad\qquad (move\_left?(\mathcal{M}, state, cell\_C)),\\
\qquad\qquad\qquad \llbracket next\_state(\ldots),\ cell\_L,\ rest\_L,\ \texttt{cons}(changed\_cell(\ldots), tape\_R)\rrbracket\\
\qquad\qquad (\texttt{t},\\
\qquad\qquad\qquad \llbracket next\_state(\ldots),\ cell\_C,\ tape\_L,\ tape\_R)\ \rrbracket)
\end{array}
$$

The first of the result expressions in the proper syntax, obeying the safety constraints (including reconstructing *tape_L* from *cell_L* and *rest_L*), is

$$
\begin{array}{l}
\texttt{cons}(\ \texttt{min}(\underline{next\_state(\mathcal{M}, state, cell\_C)}, state),\\
\qquad \texttt{cons}(\ cell\_R,\\
\qquad\qquad \texttt{cons}(\ \texttt{cons}(\ \texttt{min}(\underline{changed\_cell(\mathcal{M}, state, cell\_C)}, cell\_C),\ \texttt{cons}(cell\_L, rest\_L)),\\
\qquad\qquad\qquad \texttt{cons}(\ rest\_R,\ \mathcal{X}))))
\end{array}
$$

Recall that examinable subexpressions are identified <u>thusly</u>. The uses of `min` are effective because of the conditions that all state encodings, respectively all symbol encodings, have the same size.

6.4. *Theorem*: For every list generic Turing machine $\mathbf{M}$, there is a list $\mathcal{M}$ encoding $\mathbf{M}$ such that, for every $\ell \in \mathcal{L}$ such that $\mathbf{M}$ halts in n steps given ℓ,

$$simulate(\mathcal{M}, \ell, \hat{n}) = \texttt{string2list}(\mathbf{M}(\texttt{list2string}(\ell)))$$

Nesting and Genericity

The Turing machine simulation presented above is based on the traditional machine model with one flat tape, so it is necessary to transform an input list into a flat structure in the Turing machine's domain and transform the machine's result back into a list structure. However, Turing machines are defined on a finite alphabet while our language has an infinite set of possible symbols. The traditional method for handling an infinite alphabet is to encode symbols into strings over a binary (or other fixed) alphabet, but this begs the question of the encoding. Fortunately we are only concerned about generic computations, which means that any one-one encoding will do[8]. Thus, for each input we build an encoding specifically for that input, in one step solving the alphabet size problem and guaranteeing genericity.

The transformation of an arbitrary list structure into an encoding of the form expected by the Turing machine is a composition of four substeps. Step 2 – 4 ensure genericity by replacing each symbol with an appropriate encoding.

1. Apply the base function `flatten`.
2. Extract a list of unique symbols from the flattened list (this can be done in `RLL`).
3. Build a "symbol table" with pairs of symbols and their encodings.
4. Use the symbol table to map the flattened list.

After the Turing machine simulation, the function `unflatten` is applied to produce the actual list output. This function takes a flattened representation of a list into the actual list. The `unflatten` function can be written in `RLL`. This enables us to prove that the language $\texttt{RLL}_{SR}$ computes all generic polynomial time list functions.

6.5. *Theorem*: If $F : \mathcal{L}^n \to \mathcal{L}$ is a generic polynomial-time computable function, then there exists an `RLL` expression $e_F(x_1, \ldots, x_n)$ such that $\lambda x_1, \ldots, x_x e_F$ implements the function F.

7. Generalized Recursion

As has been noted before, the recursion scheme $\mathcal{RS}$, given in Definition 3.5, moves along a list following `rest` but not "down" a substructure following `first`. Introducing a second linear recursion, similar to $\mathcal{RS}$ but interchanging `first` and `rest`, is not sufficient because the two linear schemes cannot handle a structure requiring alternating `first` and `rest` operations for its traversal, such as that shown in Figure 2.

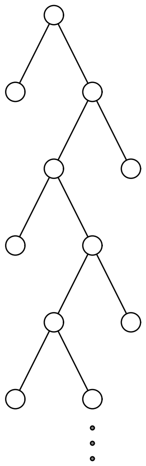

Figure 2.

A structure which cannot be traversed by linear recursions.

7.1. *Definition*: The *general recursion scheme*, denoted $\mathcal{GRS}$, defines

$$\begin{array}{lll} f \stackrel{\text{def}}{=} \lambda\, y, x : \ \texttt{cond}((\,\texttt{null?}(x), & g_1(y,\, \epsilon\,)\,), \\ \qquad\qquad\qquad (\,\texttt{atom?}(x), & g_2(y,\, x)\,), \\ \qquad\qquad\qquad (\,t, & h(y,\, x,\, f(y,\, \texttt{first}(x)),\, f(y,\, \texttt{rest}(x))))) \end{array}$$

This is written $f \stackrel{\text{def}}{=} \mathcal{GRS}[g_1, g_2, h]$. As before, y may be omitted or may be a vector $y_1, \ldots, y_k$.

7.2. *Proposition*: `flatten` is expressible using $\mathcal{GRS}$.

Proof: $\mathcal{GRS}[\mathtt{I},\ \mathtt{I},\ h]$ will suffice, where

$$h \stackrel{\text{def}}{=} \lambda y,\ x,\ w,\ z:\ \mathtt{if\text{-}then\text{-}else}(\ \mathtt{atom?}(\mathtt{first}(x),\ \mathtt{cons}(w,\ z),\ append(\ \mathtt{cons}((\epsilon),\ w),\ \mathtt{cons}((\epsilon,\ \epsilon),\ z)\ ),$$

7.3. *Definition*: A function $f \stackrel{\text{def}}{=} \mathcal{GRS}[g_1,\ g_2,\ h]$ is defined by *safe* general recursion if g_1 and g_2 are safe functions and h is a safe function returning its last two arguments.

7.4. *Definition*: The language $\mathtt{RLL}_{\text{SGRS}}$, where "SGRS" indicates safe general recursion schema, is defined analogously to $\mathtt{RLL}_{SR}$ except that $\mathcal{GRS}$ is allowed, in addition, in level–1 and above.

7.5. *Theorem*: $\mathtt{RLL}_{\text{SGRS}}$ is complete for polynomial time list computations.

The proof of proposition 4.14 then carries over directly. Proposition 4.13 requires replacing $len(x)$ with $|x|$ and a second, similar induction for $\mathcal{GRS}$.

7.6. *Theorem*: Every $\mathtt{RLL}_{\text{SGRS}}$ program runs in time polynomial in the size of its input.

Finally, we observe that $\mathcal{GRS}$ may be used to replace the primitive `min` with a more limited `scopyatom`. The restricted `scopyatom`, defined as $\lambda x, y : \mathtt{if\text{-}then\text{-}else}(\mathtt{atom?}(x) \vee \mathtt{null?}(x), x, \epsilon)$, is still necessary to check that a cell is consumed every time a cell is allocated. It would be necessary to define `min` in terms of `scopyatom` and prove that `min` is indeed safe.

Using $\mathcal{GRS}$, it is easy to do things such as XML's path expression searches. For example, seaching for all nodes of type "A" requires a function defined using $\mathcal{GRS}$ on the function

$$h \stackrel{\text{def}}{=} \lambda y, x, z_1, z_2 : \mathtt{if\text{-}then\text{-}else}(\mathtt{isType}(x, y), x, \mathtt{append}(z_1, z_2)).$$

This of course requires an appropriate representation of XML trees as lists and a corresponding definition of `isType`.

8. Conclusion

The work we have presented in this paper is very much in the spirit of programming language design, in the sense that its goal is to achieve just the right degree of capability while retaining as much perspicuity as possible.

One programming language issue is whether this work extends beyond lists to trees, DAGs, and even cyclic graphs. Grumbach and Milo [5] consider acyclic structures with great generality. Trees are handled simply by adding `atom?` and corresponding functions to the various recursion schema. DAGs provide a little more difficulty, provided we retain the same natural definition of size, which is oblivious to issues of pointer identity. The structure $(\mathcal{A},\ \mathcal{A})$, represented as a list with two copes of $\mathcal{A}$, has size $2 \times |\mathcal{A}|$. If it is represented as a DAG, with two pointers to a single structure $\mathcal{A}$, it must have the same size, which indeed it does. Thus building a DAG by replicating a pointer is potentially an unsafe operation. Restrictions on pointer replication in levels 1 and 2 could be ameliorated by an `min`-like operations that returns a pointer. This would, for example, recognize rewriting the list representation of $(\mathcal{A}, \mathcal{A})$ into the DAG representation as a safe computation. In any case, the language would remain purely functional (the only assignments are lets to local variables).

For cyclic graphs, the first difficulty is that recursion (according to the $\mathcal{RS}$ formalism or other similar schema) along a cycle will never terminate. The second is that a (naive) language for constructing cyclic graphs is no longer functional and hence a data structure may actually grow while it is being traversed. Thus substantially more work is required here.

The final question, of course, is whether this work sheds any light on the design of practical query languages for dealing with list structured data. The utility of such languages is obvious, with demands for querying information ranging from structured documents to multimedia to genome sequences. We believe that this work does contribute to the design of query languages, or at least to the systems hosting these languages. Observing the strong role that replication plays, we suggest that the system monitor the use of each input structure. Any attempt to use information for the same structure in more that one way or at more than one place in the code should cause a warning or error condition. This is, of course, much coarser than the replication control used in this paper, but it is better suited for languages which attempt to deal with collections in bulk.

Acknowledgment Latha Colby contributed a great deal to the early versions of this paper.

Bibliography

1 S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

2 V.-H. Casiero. An equational characterization of the poly-time functions on any constructor data structure. Technical Report 226, University of Oslo, Department of Informatics, 1996.

3 L. S. Colby, E. L. Robertson, L. V. Saxton, and D. Van Gucht. A query language for list-based complex objects. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 179–189, Minneapolis, MN, 1994.

4 World Wide Web Consortium. Xml query, February 2002. http://www.w3.org/XML/Query.

5 Stépahne Grumbach and Tova Milo. An algebra for pomsets. In *Proc. of the Sixth International Conference on Database Theory*, pages 191–207, Prague, Czech Republic, 1995.

6 M. Gyssens, D. Suciu, and D. Van Gucht. On polynomially bounded fixpoint constructs for nested relations. In *Proc. of the Sixth International Workshop on Database Programming Languages*, 1995.

7 M. Hoffman. Linear types and non-size-increasing polynomial time computation. In *LICS*, 1999.

8 N. Immmerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

9 D. Leivant. Stratified functional programs and computational complexity. In *POPL*, 1993.

10 Albert Meyer and Dennis Ritchie. The complexity of loop programs. In *Proceedings of the 22nd National Conference of the ACM*, pages 465–469, 1967.

11 F. Neven. On the power of walking for querying tree-structured data. In *PODS*, 2002.

12 F. Neven and T. Schwentick. Query automata. In *PODS*, 1999.

13 D. Suciu. Fixpoints and bounded fixpoints for complex objects. In *Proc. of the Fourth International Workshop on Database Programming Languages*, 1993.

14 M. Vardi. The complexity of relational query languages. In *Proc. of the Fourteenth ACM Symposium on Theory of Computing*, pages 137–146, 1982.