MIT/LCS/TM-242

EFFICIENT DEMAND-DRIVEN EVALUATION (I)

Keshav Pingali

Arvind

September 1983

Efficient Demand-driven Evaluation (I)

Keshav Pingali

Arvind

19 September 1983

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

We describe a program transformation technique for programs in a general stream language L whereby a data-driven evaluation of the transformed program performs exactly the same computation as a demand-driven evaluation of the original program. The transformational technique suggests a simple denotational characterization of demand-driven evaluation.

Keywords: Data-driven evaluation, dataflow, demand-driven evaluation, demand propagation, functional languages, lazy evaluation, least fix-points, program transformation, streams.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

Efficient Demand-driven Evaluation (I)

1 Introduction

Applicative languages give the programmer the power to construct and apply functions. In some applicative languages, the programmer is also given certain base functions like *+*, ***, *if-then-else* etc. An interpreter for such a language is capable of performing function application, and, if the language has base functions, can produce the result of the application of a base function. Applicative languages have the Church-Rosser property - *i.e.*, an interpreter for an applicative language can do the function applications in any order it chooses, because, other than termination, the outcome of the interpretation does not depend on the sequence in which the interpreter chooses to perform function applications. This pleasant property of applicative languages has generated considerable interest in both their theory and implementation.

Parallelism in a program written in an applicative language can be exploited by simultaneously evaluating all the arguments of a function application. This idea can be applied recursively if the arguments themselves are function applications, with the result that any computation can be done as soon as its inputs are available. This scheme can be loosely labeled *data-driven* evaluation. An interpreter that implements this rule of evaluation is called a *data-driven* interpreter. There are many varieties of data-driven interpreters in the literature - notably, the interpreters of Dennis [6] and Arvind *et al* [2].

It is well-known that in the presence of *non-strict* functions, a data-driven interpreter may perform some computations which are not required to produce the output of the program. A function f(x,y) is said to be *strict with respect to argument x* if the value of the function application is undefined whenever the value of *x* is undefined. If a function is strict with respect to all of its arguments, it is simply called *strict*. The function f(x,y) is a *non-strict* function of *x* if it is not strict with respect to *x*. Consider, for example, the function f(x,y) which simply returns the value of *x*[1]. Let us assume that the semantics of function

[1] This is like the K combinator of λ-calculus

applications permit a function application to be carried out before *all* the arguments of the function have been evaluated. In such a case, the function *f* is non-strict with respect to *y*, since it can return the value of *x* as soon as it is computed, even if the value of *y* is undefined. A data-driven interpreter would evaluate both *x* and *y* even though the value of *y* is not required to produce the result of the function application. If the computation of *y* terminates, then the data-driven interpreter will have done some *bounded* amount of extra computation. In most implementations, this is quite acceptable. However, if the computation of *y* does *not* terminate, then the interpreter will do an *unbounded* amount of additional computation. The reader may feel that one solution is to "kill" the computation of *y* as soon as the application of *f* returns the result, since it is clear at that point that the value of *y* is not required. However, in most multi-processors, it is very difficult to kill computations while they are executing. The ability to kill computations implies the ability to kill sub-computations spawned by this computation faster than the computation itself can generate them. Moreover, the implementation would need some efficient way of identifying all sub-computations generated by this computation. Since these sub-computations may be spread across many processors, the problem is similar to the real-time, multi-processor garbage collection problem.

We would like to emphasize that even if the data-driven interpretation does not terminate, it will still produce the results of the program (at least in theory !). Why then are we worried about non-terminating computations ? The main problem is that even in multi-processor implementations such as the Id machine [3], too many such non-terminating computations can result in the wastage of machine resources, and make the system unacceptably slow. On the other hand, it can be argued that in most situations, a good programmer never deliberately writes programs that do not terminate - if he inadvertently wrote one that did not terminate, he could not reasonably expect the machine to execute efficiently and produce the results of the program. Perhaps a plausible argument along these lines can be made when the programmer is confined to simple data-types like integers and arrays. However, the argument falls apart the moment we introduce non-strict data constructors into the language.

An example of a non-strict data constructor is the non-strict version of the LISP operator *cons*, which was introduced by Friedman and Wise [7]. The data structure *stream* which is constructed by using the non-strict *cons* has been incorporated into dataflow languages primarily to express parallelism which would have been impossible to express without it [2, 16]. When such data structures are defined in a recursive (or mutually recursive) manner, one ends up with data structures which are potentially infinite. The incorporation of infinite data structures into programming languages provides the programmer with a powerful tool for writing structured and elegant programs as anyone who has written a "sieve of Erastosthenes" program to compute the first 1000 prime numbers (as opposed to a program for computing all prime numbers between 1 and 1000 !) can confirm. Further examples of such programs can be found in [5, 8, 14]. Once we permit (and encourage) the programmer to use non-strict constructors to define infinite data-structures, it would appear that data-driven computation would be extremely inefficient. Although a data-structure may be infinite by definition, it is usually the case that only a few elements of the data-structure are required to produce the output of the program. A data-driven interpreter will attempt to compute all the elements of the data structure, and is, therefore, not a practical method for implementing such data structures.

For these reasons, some implementors of applicative languages have preferred an alternative paradigm of interpretation called *demand-driven* interpretation. A demand-driven interpreter is normally thought of as performing precisely those computations that are required to produce the output of the program. Typically, a request for printing the value of some variable in a program is considered to be a demand for the value of that variable. The computation of this value will require, in general, the values of other variables in the program. The process of identifying those variables whose values are required to produce the demanded output is commonly called *demand propagation*. Since a demand-driven interpreter performs only those computations which are required to produce whatever output has been demanded, it will, in general, perform less computation than a data-driven interpreter. However, a demand-driven interpreter has to propagate one demand for each data value that is computed; the overhead of demand propagation is not

present in data-flow interpreters. If a language has only *strict* functions (like FP [4]), then a demand-driven interpreter for the language is unattractive.

It is also the case that many programs show greater parallelism under data-driven evaluation than they do under demand-driven evaluation. Typical of such programs is the producer-consumer scenario. If a producer is allowed to compute values only when they are demanded by the consumer, pipe-lined operation of the producer and consumer is often ruled out. Some researchers have attempted to fix this problem by permitting the producer to go ahead of the consumer by some fixed amount [11, 12]. Besides seeming somewhat *ad hoc*, such schemes still have the problem that they must propagate as many demands as an ordinary demand-driven interpreter would. In fact, data-driven evaluation of some stream programs is modeled by Keller *et al* [12] by essentially issuing an infinite number of demands !

In this paper, we approach the problem of combining data-driven and demand-driven evaluation in a very different way. Starting with a data-driven interpreter, we show how the effect of demand-driven evaluation can be achieved through *program transformation*. First, we define a general stream processing language L. We then give a program transformation technique for programs in L such that a data-driven evaluation of the transformed program computes precisely the same data values as a demand-driven evaluation of the original program. This transformation is not a source-to-source transformation; rather, it transforms L programs into programs in a language which is a super-set of L and which we call the language LD. We will refer to the LD programs resulting from this transformation as *lazy programs*. We show that a data-driven evaluation of lazy programs satisfies four properties named P1 to P4, which are also satisfied by a demand-driven evaluation of the original L program. Although we do not present it as such, these properties suggest a simple denotational characterization in the style of Kahn [10] of demand-driven interpretation.

In a companion paper, we show that the set of programs that can be expressed in LD include not only L programs and lazy programs, but also programs that can best be

described as *partially* demand-driven. In particular, we can define programs that are input-output equivalent to lazy programs but which do not necessarily compute minimal histories on internal lines. We will use this idea to present a transformation which trades the complexity of demand propagation for a *bounded* amount of extra computation on some data lines. We refer the reader to the companion paper for more details.

2 Demand Propagation in a General Stream Language

In this section, we introduce a general stream language L. We will use L programs to illustrate our technique for introducing demand streams into programs in an applicative language. This section lays down the framework for some program optimizations which we will discuss in later sections.

2.1 L - a General Stream Language

Language L has two classes of data- *scalar values* and *streams*. Scalar values encompass the usual data types like integers, reals, booleans, character strings etc. Streams are sequences of scalar values and are constructed by using a non-strict data constructor. For the purpose of our discussion, we will represent streams as $[a_1, a_2, ...]$ where a_1 is a scalar value that is the first element of the stream, a_2 is a scalar value that is the second element of the stream etc. The empty stream (*i.e.*, the undefined stream) is represented as [ ]. It is possible to introduce a special scalar value *est* (*i.e.*, end-of-stream) and let an empty stream be the stream containing exactly one scalar value *est* [16]. There are no difficulties in extending L to include such a feature, but we will not do so in this paper.

The usual stream operators *first*, *rest* and *cons* are included in L. Their functionality is summarized below -

$first([\,]) = \perp_s$ (the undefined scalar value)
$first([a_1, a_2, a_3, ...]) = a_1$

$rest([\,]) = [\,]$
$rest([a_1, a_2, a_3, ...]) = [a_2, a_3, ...]$

$cons(\perp_s, [a_1, a_2, ...]) = [\,]$
$cons(b_1, [a_1, a_2, a_3, ...]) = [b_1, a_1, a_2, a_3, ...]$

Since [ ] is the undefined stream, there are no operators that can test a stream for emptiness. Notice that since $\perp_s$ is the *scalar* undefined value, its "type" is different from that of [ ], and hence, we use a different symbol for it.

In addition to these operators, L has any *strict and total function* like +, *, etc. which takes some scalar inputs and produce a scalar output. Such functions are called scalar functions and are represented by t, or t with a subscript. L also admits stream versions of these functions which are defined in the following way. Let *t* be a scalar function of *n* scalar arguments. The stream version of *t*, represented by *T*, is a function of n streams (say A, B, C, ...) and produces an output stream O such that O(i), the i^{th} element of O, is t(A(i), B(i), C(i), ...). We will refer to such stream operators as *T-boxes*. Since T-boxes operate "point-wise" on their inputs, they have the "one-in-one-out" property - *i.e.*, to produce *k* elements of the output stream of a T-box, we need *k* elements of all input streams of the T-box. The reader may wonder why we have both t-boxes and T-boxes in the language. There is no deep reason behind this - permitting t-boxes in the language simplifies the discussion in a companion paper, and hence, is a notational convenience.

Although a large class of stream programs can be written using the operators described above (for example, programs to compute Fibonacci numbers and factorials !), a useful stream language must have some ability to manipulate streams in a manner other than a one-in-one-out way. Three very useful operators that are *not* T-boxes are *true-gate*, *false-gate* and *merge* whose operational behavior is described below -

- *true-gate*(B, X) - B is a stream of booleans. X(i) is output if B(i) is *true*; otherwise, it is absorbed. In other words, the j^{th} element of the output stream is X(i) if B(i) is *true* and the number of *true* values between B(1) and B(i) is j.

- *false-gate*(B, X) - Its behavior is exactly like that of a true-gate, except that X(i) is output if B(i) is *false*.

- *merge*(B,X,Y) - B is a stream of booleans. The i^{th} token on the output stream is X(j) if B(i) is *true* and the number of *true* tokens between B(1) and B(i) is j, and Y(k) if B(i) is *false* and the number of *false* tokens between B(1) and B(i) is k.

Examples of the use of these operators can be found in papers by Weng [16] and Turner [14].

Language L includes *true-gate*, *false-gate* and *merge* operators.

An L program is a set of recursive definitions where the left hand side (LHS) of each definition consists either of a *scalar variable* or a *stream variable*. The right hand side (RHS) of a definition consists of a function application where the function is one of the operators described above and the arguments are variables. Definitions must be *type consistent* - *e.g.*, the definition of a scalar variable must be the application of a *t* operator or a *first* operator, while the definition of a stream variable must be the application of a *rest*, *cons*, *true-gate*, *false-gate*, *merge* or a T operator.

If t-boxes and T-boxes are left uninterpreted, then the set of equations of an L program can be considered to be a partially interpreted equational scheme. Jaffe [9] has shown that the expressive power of such an equational scheme is equivalent to that of recursively enumerable schemes. Interestingly enough, the expressive power of such a scheme remains unchanged even if t-boxes and T-boxes are removed from the system. Of course, such a system would not be particularly convenient for programming.

We will find it convenient to consider an L program as a data-flow graph. The data-flow graph corresponding to an L program can be generated by drawing a box for each equation in the program, labeling the box with the function on the RHS of the equation, labeling the output of the box by the variable on the LHS of the definition and connecting the output of the box to the appropriate inputs of all boxes where it is needed. Since the output of a box may be connected to the inputs of several boxes, there is an implicit *fork* operator at the output of any box that is connected to several boxes. It is convenient to think of a scalar variable as a single token and a stream variable as a sequence of tokens flowing down the arc with the label of that variable[2]. Each out-going arc of a fork receives a copy of a token

[2]We assume unbounded buffering along each arc.

at the in-coming arc. Thus, there is a direct correspondence between the *history* of a line X in the data-flow graph and stream X in the L program. In the discussion below, we will drop the distinction between the L program and its data-flow graph, as well as the distinction between stream X in the L program and the *history* of the line labeled X in the data-flow graph, and use these terms interchangeably. Figure 1 shows the operators of L. We make *fork* an operator in order to simplify the discussion below. A program for generating Fibonacci numbers is shown in Figure 2.

As in Kahn [10], the meaning of L programs can be given as follows. If D is some set, let D^{ω} be the set of finite and denumerably infinite sequences of elements of D, where the empty sequence [ ] is considered to be an element of D^{ω}. Let $\mathfrak{D}$ be the set containing the denotations of all scalar values such as integers, booleans etc. Consider the set $\mathfrak{D}^{\omega}$. Elements in $\mathfrak{D}^{\omega}$ are ordered by the prefix ordering on sequences in which the smallest element is [ ]. It is easy to show that under this ordering, all the operators in L are monotonic and continuous functions from sequences to sequences. For each equation in an L program, we can write down a semantic equation that describes the relation between its inputs and outputs. The meaning of an L program is the least fix-point of this set of semantic equations.

2.2 Data-Driven Evaluation of L programs

Consider the following L program where F1, F2, ... represent L operators, and I1, I2, ... represent input streams.

X1 = F1(I1,I2,...,X1,X2,...)
X2 = F2(I1,I2,...,X1,X2,...)
.
.
XN = FN(I1,I2,...,X1,X2,...)

By Kleene's theorem [13], the least fix-point can be computed using the following iterative process in which $XI^{(k)}$ represents the value of stream XI at step k.

1. Substitute the value of input streams on the RHS, and set $XI^{(0)}$ = [ ] for all streams.

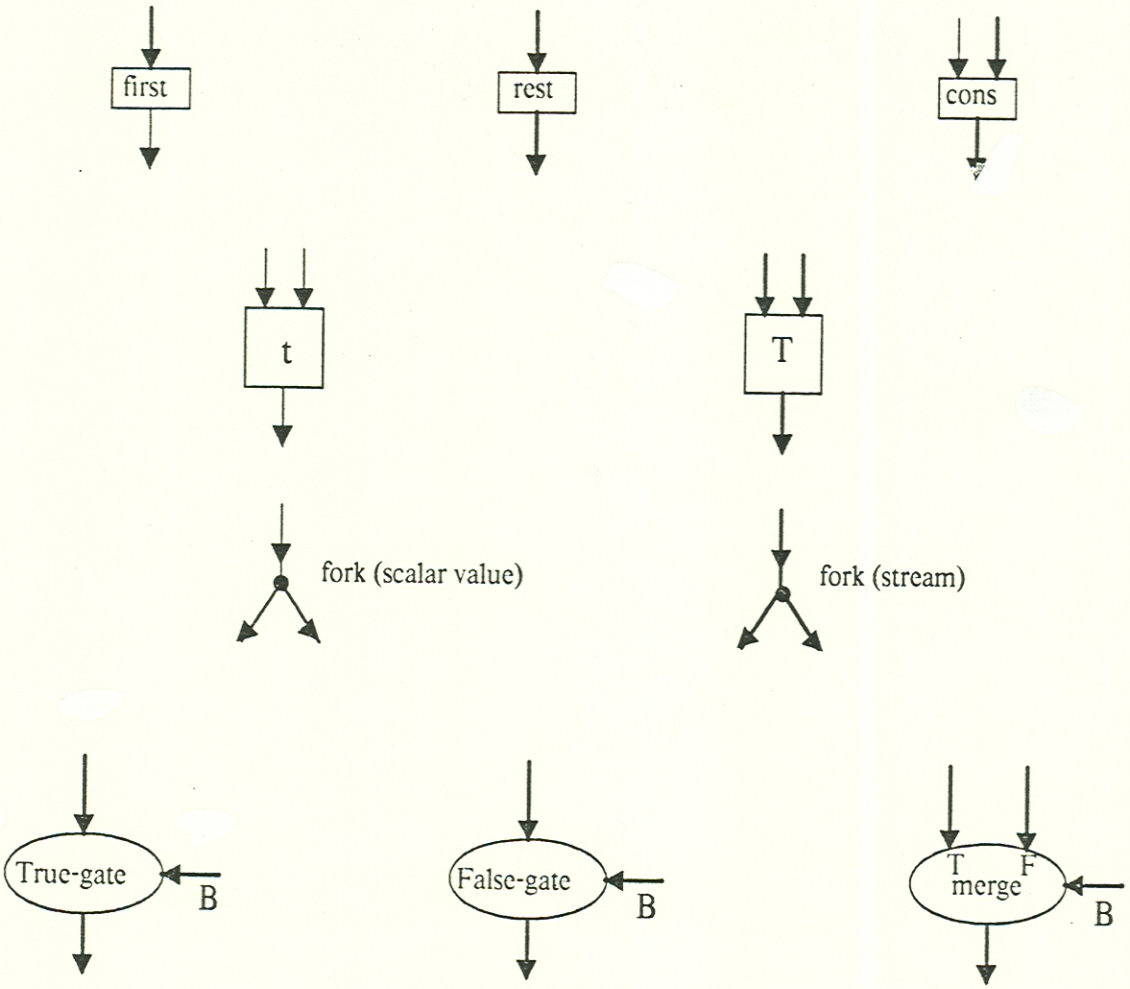


Figure 1: The Operators of Language L

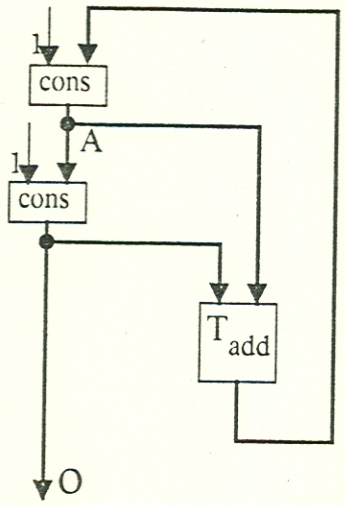


Figure 2: A Program for Computing Fibonacci Numbers

2. Repeatedly compute, $XI^{(k+1)} = FI(I1,I2,...,X1^{(k)},X2^{(k)},...)$ for all streams.

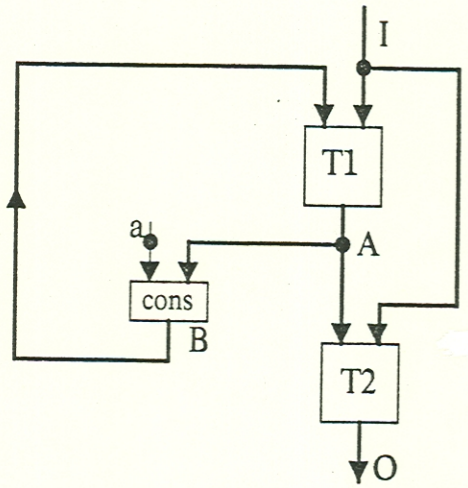
For a terminating program the iterative process will stop at some step k where $XI^{(k+1)} = XI^{(k)}$ for all streams. Several points should be noted about this process. First of all, $X^{(k)} \subseteq X^{(k+1)}$ because F's are monotonic. Therefore at each "step" old elements of X are left unchanged and a few additional elements of X may be defined (*i.e.* computed). If streams are viewed as a sequence of tokens then this process can be understood as follows: arrival of a token at the input of a function may cause a new token to be generated at the output stream. This is an essential property of data-driven or dataflow systems. However, there is a subtle difference between the technique outlined above and data-driven evaluation.

In Kleene's recursion there is a well defined concept of step. The $k+1^{st}$ step can begin only after the $X^{(k)}$ value of all streams have been computed. This is a form of *fair scheduling* which guarantees that every stream function gets a chance to be evaluated. Without this notion of fair scheduling, it can not be ensured that the least fix-point of the equations would be computed. In dataflow systems there is no precise concept of step. Data-driven evaluation may be defined as follows:

1. Substitute the value of input streams I1,I2,... on the RHS and set $XI^{(0)} = [\,]$ for all streams.

2. Evaluate any one F that has received additional input, that is, compute $X^{(k+1)} = F(I1,I2,...,X1^{(k)},X2^{(k)},...)$ such that for some Xi, $Xi^{(k-1)} \subset Xi^{(k)}$. Repeat step 2.

The above rule is nondeterministic in the sense that there are many permissible execution sequences. It is appealing for parallel machines because *any subset of enabled operators can be chosen to execute.* However it has the defect that some enabled operator may never be chosen and thus the least fix point may never be reached. This problem arises only when some stream other than the output stream can have infinite tokens. An example of such a program is shown in Figure 3.



A = T1(B, I)

B = *cons*(a, A)

O = T2(A, I)

Figure 3: A Simple L Program

2.3 Introducing Demand Streams In L Programs

We now describe our technique for transforming L programs such that a data-driven evaluation of the transformed program computes precisely the same values as a demand-driven evaluation of the original program. The technique essentially involves the introduction of demand streams into L programs. A *demand stream* is a stream of *d* tokens (for *d*emand tokens) where the data type of d is distinguished from all other data types in L such as integers, booleans, reals etc. Let stream *X* be the output of some operator A. We will represent the demand for elements of stream X by a demand stream DX, in which the i^{th} d token represents the demand for the i^{th} element of X. In order to convert demands for the output of some operator into demands for the inputs of the operator, we can associate some additional code with every operator, which will take as input the demand stream DX (and, perhaps, the inputs of the operator) and generate demand streams for each of the inputs. For scalar operators, we will assume that only one demand for the output can be made, and this demand will be represented by the scalar variable d. The code for demand

propagation for each operator of L is shown in Figure 4. Notice that except in the case of the *fork* operator which requires the D-union operator, no new operators are required for demand propagation.

The code for a *fork* needs some explanation. A *fork* has two outputs, and hence, the demand streams for the outputs must be merged together to generate one demand stream for the input of the *fork*. The operators d-union and D-union perform this operation for scalars and streams respectively. Their behavior is easy to understand in operational terms. The *d-union* operator can receive zero or one d token on either of its input arcs. When the *d-union* operator receives the first d token, it forwards it to the demand stream for the input stream of the *fork*. If, after this, it receives a d token on its other input arc, it simply absorbs this token. Since a *d-union* operator does not know *a priori* on which input arc it will first receive a d token, the implementation of the *d-union* operator requires some concept of nondeterminism. However, the *d-union* operator is still a monotonic and continuous function from histories to histories - given the histories of its input lines, the history of the output of the *d-union* is uniquely defined. An alternative way of looking at the d-union operator is that it is the *least upper bound* operator on a domain in which there are only two elements - $\perp_s$ and d.

D-union is the stream version of *d-union* which forwards only one of the two i^{th} tokens to arrive on inputs. For example, if tokens on the input lines A and B of a *D-union* arrive in the time order - A(1), A(2), B(1), B(2), B(3), ... then its output will be A(1), A(2), B(3), ... The *D-union* operator is also a function from histories to histories.

Since we do not want to perform any computation that is not done by a demand-driven evaluator, a token should not be permitted to flow down the right branch of the fork if it was generated in response to a demand from the left branch of the fork (and vice versa), until there is a demand for the token from that branch. Operators *gate* and *GATE* permit us to do this for scalar and stream *forks* as shown in Figure 4(f) and Figure 4(g).If *d* is a demand, *gate*(x,*d*) is a strict scalar function which outputs x only when both x and *d* have

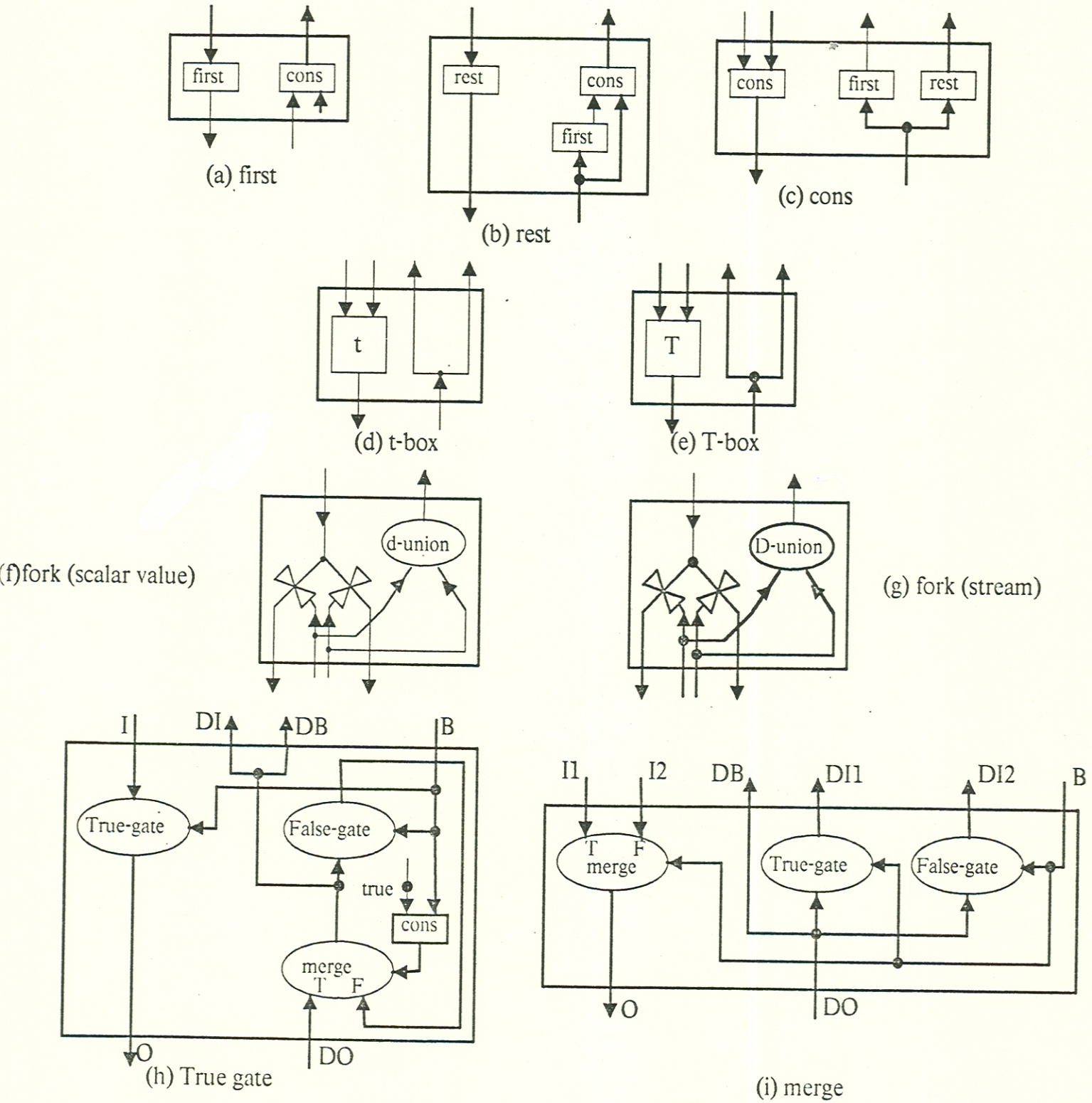


Figure 4: Propagating Demands

been received. *GATE* is simply the stream version of *gate* and hence, is a T-box. *Gate* operators are shown as  in Figure 4.

Once we can associate code for demand propagation with each operator of L, it is easy to introduce demand streams into L programs - simply convert each operator in the L program into the operator with its demand propagation code as specified by Figure 4. We will describe the algorithm more formally in terms of the dataflow graph associated with the L program.

Algorithm-MDP: A Microscopic Demand Propagation Algorithm for introducing demand streams into L programs.

Let F be any operator in an L program, and let its inputs be I1, I2, ... and let its outputs be O1, O2, ... Let FD denote operator F with its demand propagation code. In FD, let DO1, DO2, ... denote the demands for outputs O1, O2, ... respectively and let DI1, DI2, ... denote the demands for inputs I1, I2, ... respectively. Note that DO1, DO2, ... are *inputs* of FD and DI1, DI2, ... are *outputs* of FD. Given the dataflow graph of an L program, the *LD program (i.e., the L program with Demands)* that corresponds to this L program is generated as follows -

1. transform each operator in the L program into the operator with its corresponding demand propagation code as in Figure 4

2. let us call each line in the L program a *data-line*. For each data-line A, do the following :

   - if A is output On of operator F and input Im of operator G, then create

     a. a line labeled A in the LD program, and let it be output On of FD and input Im of GD.
     b. a line labeled DA in the LD program, and let it be output DIm of GD and input DOn of FD.

   - if A is an *input line* of the program and is input Im of operator F, then create

     a. an input line labeled A in the LD program, and let it be input Im of operator FD.
     b. an *output line* labeled DA, and let it be output DIm of operator FD.

- if A is an *output line* of the program and is output Om of operator F, then create

  a. an output line labeled A in the LD program, and let it be output Om of operator FD.
  b. an *input line* labeled DA, and let it be input DOm of operator FD.

□

Notice that for every line in the L program, there is a line in the LD program with the same label. The LD program corresponding to the L program shown in Figure 2 is shown in Figure 5.

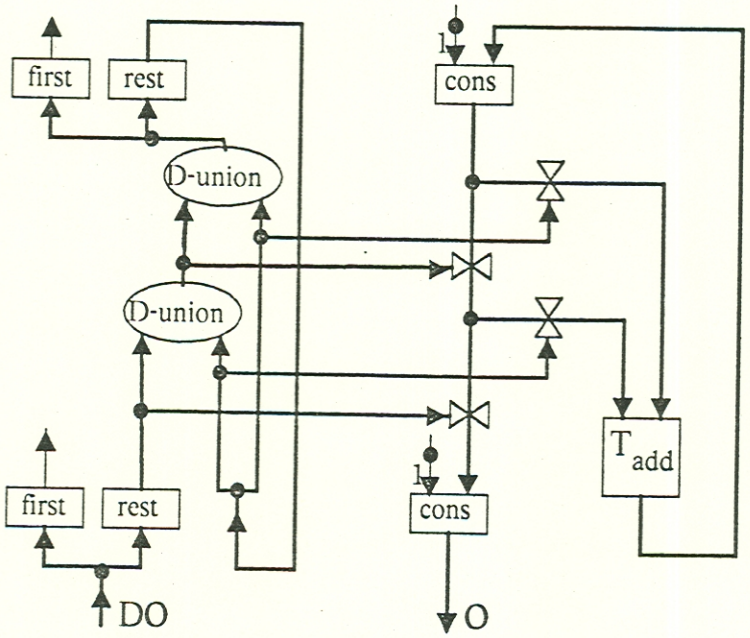


Figure 5: The LD program corresponding to the program of Figure 2

We now give the semantics of LD programs. Let d be the denotation of d, the demand data type. d is incomparable to all other data types such as integers, reals, booleans etc. Let $\{d\}^\omega$ be the set of all finite and denumerably infinite sequences of d's. Construct a domain $\mathfrak{D}om$ which is the union of $\mathfrak{D}^\omega$ and $\{d\}^\omega$. As before, these sequences can be ordered by the prefix ordering on sequences. It is easy to show that in this domain, all the operators of LD are monotonic and continuous functions from sequences to sequences - the only new operators are d-union and D-union whose semantic equations are

d-union(a, b) => *if* a = $\perp_s$ *then* b *else* a

D-union(A, B) => *cons*(d-union(*first*(A), *first*(B)),
D-union(*rest*(A), *rest*(B)))

where a,b $\in$ {d, $\perp_s$} and A,B $\in$ {d^*}

Given an LD program, we can write down a set of semantic equations that express the relations between input and output streams of each operator in LD. The meaning of LD programs is the least fix-point of this set of semantic equations.

2.4 Data-driven Evaluation of LD Programs

Since all the operators in LD programs are monotonic and continuous stream functions, LD programs can be executed on a dataflow machine which executes L programs. Notice that the execution of LD programs is done in a data-driven manner - *i.e.*, any stream operator that receives input can be executed. In the next subsection, we will show that data-driven evaluation of an LD program will compute precisely the same histories on every line as a demand-driven evaluation of the corresponding L program. Consequently, *no* fair-scheduling will be required for the execution of LD programs as long as the computation of the output of the program does not require infinite histories on any line. Of course, once an infinite history on some line is required to produce the output of the program, there must be fair-scheduling to produce the required output on *all* output lines. This is not a draw-back of our scheme - any implementation of a demand-driven evaluator will require fair-scheduling in such a case.

2.5 Properties of LD programs

In subsections 2.1 and 2.4, we have shown that the meaning of L and LD programs can be expressed in the context of Kahn's theory. An important benefit of doing this is that we can use techniques like stepwise computational induction [13] and induction on the structure of programs to prove properties of L programs. In this subsection, we will use these techniques to prove that our method of introducing demand streams into L programs is "correct" in the sense that a data-driven evaluation of the LD program will produce the same results as a demand-driven evaluation of the L program.

Since demands are represented explicitly in LD programs, it is reasonable to expect that the criteria for "demand-drivenness" should be expressible in the context of the semantics of LD programs. Of course, the fact that we call some line in an LD program a "demand line" does not automatically mean that the history of that line has anything to do with demands in a demand-driven evaluator - for example, if the code for demand propagation through some operator is incorrect, or if our operator-by-operator transformation of the L program is incorrect, then the histories of demand lines in LD programs may have no relationship with the histories of the corresponding data lines in the L program under a demand-driven interpretation. In fact, the proof given below will, in some sense, provide an *a posteriori* justification for calling these lines demand lines !

The proof we give below relies on a comparison of the computations performed by an L program and the corresponding LD program when they are both given the same inputs[3]. We now introduce some notation. If A is the name of some line in an L program, let HA and LA denote the histories of the lines labeled A in the L and LD programs respectively at any point in the computation, and let LDA denote the history of line LDA (*i.e.*, the demand line corresponding to A in the LD program) at any point in the computation. We let $\mathfrak{H}$A, $\pounds$A, and $\pounds$DA denote the "final" histories of these lines - *i.e.*, the histories defined by the least fix-point of the system of equations for the L and LD programs. Let *Data-lines* stand for the set of lines in the L program, *In* stand for the subset of Data-lines which is the set of all *input* lines of the L program and *Out* stand for the subset of Data-lines which is the set of all *output* lines of the L program. If X is some history, we will let $|X|$ denote the length of the history where length is the (obvious) function defined on histories. We would like to emphasize that *length* **is not an operator in the language**, and hence, is *not* a function from a stream to an integer; rather, it is a function from the denotation of a stream to the denotation of an integer.

Before we prove the main result of this section, we attempt to give some intuition behind

---

[3] *i.e.*, the same *data* inputs. The LD program will need some *demand* inputs as well. It is useful to imagine an L program and the corresponding LD program placed side-by-side and given the same data inputs.

it. We would like to prove that a data-driven evaluation of an LD program will perform the same computations as a demand-driven evaluation of the corresponding L program. How would one characterize "demand-driven evaluation of the L program"? The first property we would expect of a demand-driven evaluation of an L program is that the history of any line X in the program will be a prefix of the history of the corresponding line X in the L program under *data-driven* evaluation. In other words, under demand-driven evaluation, fewer tokens may flow along a data-line in the program - however, the *values* of these tokens will be the same under both modes of evaluation. In our model, this property can be stated more formally as follows -

$$\wedge_{X \in \text{Data-lines}} \{\mathfrak{L}X \subseteq \mathfrak{H}X\}$$

A second property we would expect our system to have is that for any data-line X, $|\mathfrak{L}X| \leq |\mathfrak{L}DX|$. In our model, we would like to identify the number of elements demanded of a stream with the number of d tokens on the corresponding demand-line - this identification would be meaningless if there could be more tokens produced on a data-line than the number of d tokens produced on the corresponding demand-line. A data-line for which this property is true will be said to be *constrained* by its demand-line. Although this property will be true for all lines at all points of the computation, we will need only the weaker result that the property is true for the "final" histories on all lines. This property can be stated more formally as follows -

$$\wedge_{X \in \text{Data-lines}} \{\ |\mathfrak{L}X| \leq |\mathfrak{L}DX|\ \}.$$

If X is a data-line and $|\mathfrak{L}X| < |\mathfrak{L}DX|$, then we will call line X *unsatisfied.*

We can now formalize the intuitive notion of demand-driven evaluation as being that form of evaluation which computes the smallest histories on all lines which suffice to produce the output of the program. In our model, we can express this notion by saying that if any data-line is unsatisfied, then there must be some output data-line that is unsatisfied. More formally, this can be expressed as follows -

$$\wedge_{O \in Out} \{ |\mathfrak{L}O| = |\mathfrak{L}DO| \} \Rightarrow \wedge_{X \in Data\text{-}lines} \{ |\mathfrak{L}X| = |\mathfrak{L}DX| \}$$

Notice that an "interpreter" that performs *no* computation whatever would also satisfy all three properties ! In order to rule this out, we need a property that avers that the interpreter does, in fact, do some work. The appropriate property is that under demand-driven evaluation, if some data-line is unsatisfied, then the history of the data-line is the same as the history of the line under data-driven evaluation. More formally, we can express this property as follows -

$$\wedge_{X \in Data\text{-}lines} \{ |\mathfrak{L}X| < |\mathfrak{L}DX| \Rightarrow |\mathfrak{L}X| = |\mathfrak{H}X| \}$$

The reader should convince himself that the four properties given above formally capture the notion of demand-driven evaluation. In the rest of the subsection, we will prove that data-driven evaluation of LD programs has the four properties given above by showing that any program built out of operators that satisfy the four properties also satisfies the four properties.

Since the set of input lines is a subset of the set of data-lines of the LD program, the four properties given above imply certain constraints on how inputs are to be fed into the LD program. For example, suppose that *A* is an input line, and suppose that the user has provided 5 data values to be fed into line A. Suppose, however, that only 3 data values are required on this line to produce whatever output has been demanded by the user. In that case, the second property requires that only 3 data values be allowed to flow down line A. How is this to be achieved ? An LD program has two kinds of inputs - data inputs and demand inputs. Providing demand inputs is easy - if *n* values are required to be produced on output line *Om* of the LD program, then *n* d tokens are fed into the input line *DOm*. The demand propagation code of the LD program propagates these demands through the LD program and generates demands for the *inputs* of the LD program. Let *I* be an input line, and suppose that *n* tokens are required on that line to produce whatever output has been demanded from the LD program. If demands have been propagated correctly, then $|\mathfrak{L}DI|$ will be equal to *n*. However, the user may have supplied fewer than *n* tokens, or *n*

tokens, or more than *n* tokens on line I. In order for the properties to hold, it is necessary that if the user has supplied *n* or more tokens to be fed into line I, then exactly *n* tokens must be fed into line I; on the other hand, if the user has supplied *fewer* than *n* tokens on line I, then *all* these tokens must be fed into the program. This requirement on input lines can be expressed formally as follows -

$$\wedge_{I \in In} \{ (\mathfrak{L}I \subseteq \mathfrak{H}I) \wedge (|\mathfrak{L}I| \leq |\mathfrak{L}DI|) \wedge (|\mathfrak{L}I| < |\mathfrak{L}DI| \Rightarrow |\mathfrak{L}I| = |\mathfrak{H}I|) \}.$$

For obvious reasons, we will say that inputs are fed into an LD program *on demand* if the requirement given above is satisfied. In the rest of this sub-section, we will show that if the condition given above for input lines is satisfied, then a data-driven evaluation of LD programs will satisfy the four properties given earlier. Exactly how the requirement on input lines is ensured operationally is an implementation dependent issue. For example, if the generation of a d token on the demand line corresponding to some input line causes the system to prompt the user for one data token or causes it to read the required token from a file, then it is ensured that tokens are fed into input lines only when there is demand for them. On the other hand, if the implementation requires that all tokens that are to be fed into input lines must be placed on the lines before the execution of the program (*i.e.*, if the input lines must be initialized with the input histories), then we must put a gate on each input line and control the gate with the demand line for that input, thereby ensuring that input tokens are allowed to enter the LD program only on demand.

We will now prove that LD programs satisfy the four properties mentioned earlier. It will be convenient to prove slightly stronger versions of the four properties in which the conditions on input lines are brought in explicitly. Thus, the properties may be restated as follows.

P1(Correctness) :A demand-driven evaluation of a program never performs more computation than a data-driven evaluation of the program.

$$\wedge_{I \in In} \{ \mathfrak{L}I \subseteq \mathfrak{H}I \} \Rightarrow \wedge_{X \in \text{Data-lines}} \{ \mathfrak{L}X \subseteq \mathfrak{H}X \}$$

P2(Demand-driven Lines): If all input lines are constrained by their demand-lines, then all data-lines are constrained by their demand-lines.

$$\wedge_{I \in In} \{ (\mathfrak{L}I \subseteq \mathfrak{H}I) \wedge (|\mathfrak{L}I| \leq |\mathfrak{L}DI|) \} \Rightarrow \wedge_{X \in \text{Data-lines}} \{ |\mathfrak{L}X| \leq |\mathfrak{L}DX| \}$$

P3(Liveness): If inputs are fed in on demand into the LD program, then the history of any unsatisfied data-line in the LD program will be the same as the history of the corresponding data-line in the L program.

$$\wedge_{I \in In} \{ (\mathfrak{L}I \subseteq \mathfrak{H}I) \wedge (|\mathfrak{L}I| \leq |\mathfrak{L}DI|) \wedge (|\mathfrak{L}I| < |\mathfrak{L}DI| \Rightarrow |\mathfrak{L}I| = |\mathfrak{H}I|) \}$$
$$\Rightarrow$$
$$\wedge_{X \in \text{Data-lines}} \{ |\mathfrak{L}X| < |\mathfrak{L}DX| \Rightarrow |\mathfrak{L}X| = |\mathfrak{H}X| \}$$

P4(Parsimony): If inputs are fed in on demand, and all output data-lines of the LD program are satisfied, then all data-lines in the LD program are satisfied.

$$\wedge_{I \in In} \{ (\mathfrak{L}I \subseteq \mathfrak{H}I) \wedge (|\mathfrak{L}I| \leq |\mathfrak{L}DI|) \wedge (|\mathfrak{L}I| < |\mathfrak{L}DI| \Rightarrow |\mathfrak{L}I| = |\mathfrak{H}I|) \}$$
$$\Rightarrow$$
$$[\wedge_{O \in Out} \{ |\mathfrak{L}O| = |\mathfrak{L}DO| \} \Rightarrow \wedge_{X \in \text{Data-lines}} \{ |\mathfrak{L}X| = |\mathfrak{L}DX| \}]$$

Lemma 1: All operators of LD satisfy properties P1 to P4.

Proof: Follows trivially from the semantic equations for each of the operators of LD.

□

The operators in L are powerful enough that L is a very general programming language - as we stated before, the expressive power of L schemes is the same as that of recursively enumerable schemes. However, the reader may wonder if the operators of L have been chosen carefully so that properties P1 to P4 hold, or whether the properties hold for a language which is augmented by other operators. One might feel that as long as every operator in the language is a monotonic and continuous function (from histories to histories), properties P1 to P4 should hold for programs in the language. Unfortunately, this is not true - if the language has *non-sequential* functions [15], then property P4 may not necessarily hold. Consider, for example, the *parallel-or* operator shown below.

A demand for the output of a parallel-or function must be propagated to both of its

	⊥	F	T
⊥	⊥	⊥	T
F	⊥	F	T
T	T	T	T

Figure 6: Truth-table for the parallel-or

inputs since the interpreter has no way of knowing *a priori* which of the inputs may be undefined. However, since the parallel-or can produce its output even if one of its inputs is undefined, the parallel-or (together with its demand propagation code) does not satisfy property P4. We are not sure how to characterize demand-driven evaluation in the presence of non-sequential functions. However, since most programming languages do not include non-sequential operators such as the parallel-or, this omission is not very serious. Note that although the d-union operator is a non-sequential function, it is *not* an operator of L, but only of LD.

We will now show that any program built up from any operators satisfying properties P1 to P4 (not necessarily only those in LD) also satisfies these properties. It is easy to show that any L (or LD) program can be built from primitive operators by repeated use of two operations - juxtaposition and iteration, as in Figure 7.

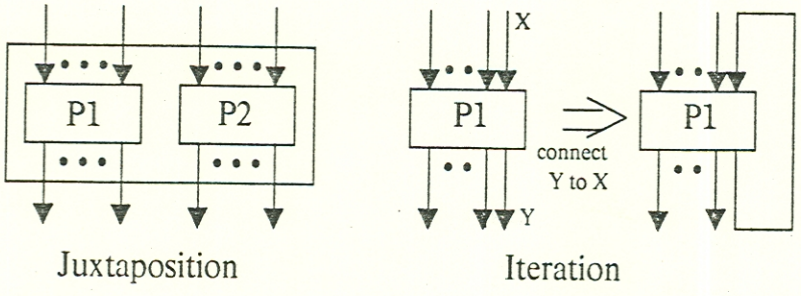


Figure 7: Juxtaposition and Iteration of Programs

Theorem 2: Properties P1 to P4 are invariant under juxtaposition and iteration.

Proof: To prove this theorem, we will use stepwise computational induction

and induction on the structure of the program. The reader who is unfamiliar with these techniques can refer to [13].

Proof of Property 1:(proof by stepwise computational induction)

Let us define a predicate $\mathcal{P}$ as follows :

$\mathcal{P} = \wedge_{X \in \text{data-lines}}[LX \subseteq HX]$

Since there are a finite number of lines in an L program, this is an admissible predicate [13].

If $LX = [\,]$ for all data-lines X in the transformed program, the predicate is trivially true.

Suppose that the histories of data-lines A, B, C, ... are *LA, LB, LC* ... in the transformed L program, and *HA, HB, HC,* ... in the L program, and suppose that the predicate is true for these histories. Each data-line in the L program is either an input line or the output of some operator. If X is an input line, then, by assumption about input lines, $LX \subseteq HX$ is always true. If X is not an input line, then it is an output of some operator (say F). Let P, Q, R, ... be the inputs to F, and let the outputs of F be labeled X, Y, Z, ... as in Figure 8.

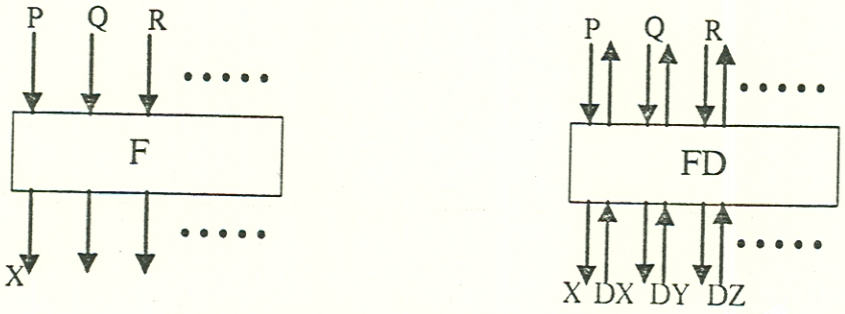


Figure 8: Operators F and FD

For the lines labeled X,

FD-x(*LP, LQ, LR, ... , LDX, LDY, LDZ, ...*)
$\subseteq$ F-x(*LP, LQ, LR, ...*) {FD satisfies property P1}
$\subseteq$ F-x(*HP, HQ, HR,....*) {LP $\subseteq$ HP, ...}

A similar argument holds for all other lines in the programs. By Scott's induction rule, we can now conclude that

$\wedge_{X \in \text{data-lines}}[\pounds X \subseteq \mathfrak{H}X]$.

□

Proof of Property 2:(by stepwise computational induction)

In this case, the (admissible) predicate is

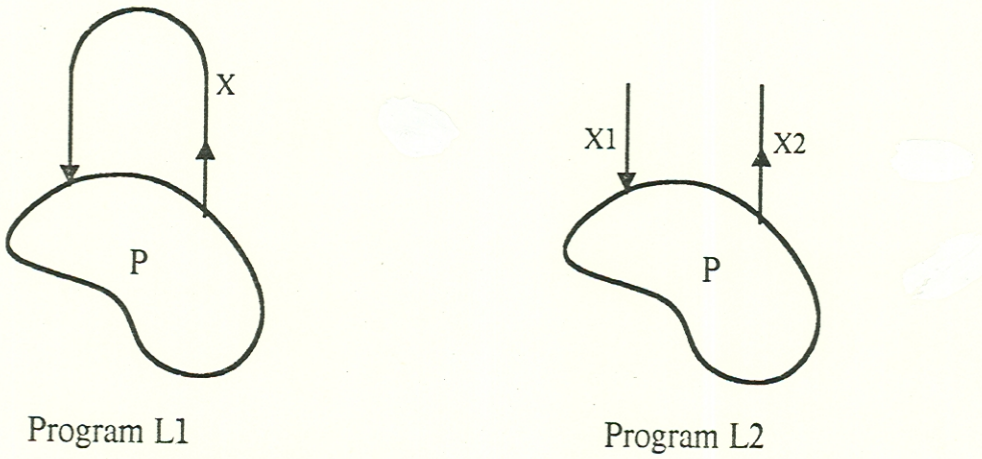
$\wedge_{X \in \text{data-lines}}[|LX| \leq |LDX|]$.

The proof follows the same lines as the proof of (1) and is, therefore, omitted.

□

We now state two simple facts about L and LD programs which are needed to prove that properties P3 and P4 are invariant under juxtaposition and iteration.

Lemma 3: Consider the two L programs L1 and L2 shown in Figure 9. If $\mathfrak{H}X1 = \mathfrak{H}X$, then $\mathfrak{H}X2 = \mathfrak{H}X$.



Program L1 Program L2

Figure 9: Cutting a line in an L program

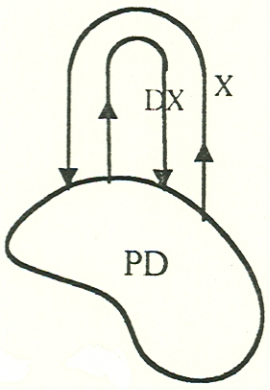
Proof: - Follows trivially from the functionality of program P.

□

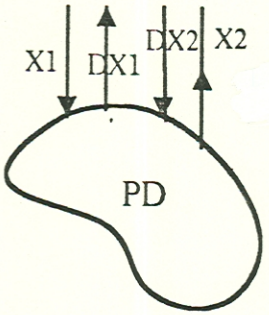
A similar lemma exists for LD programs as well - the lemma is slightly different because care must be taken with regard to data inputs to LD programs.

Lemma 4: Consider the two programs LD1 and LD2 shown in Figure 10. If $\pounds DX2 = \pounds DX$, and $\pounds X$ is fed into X1 *on demand* then $\pounds X1 = \pounds X2 = \pounds X$ and $\pounds DX1 = \pounds DX2 = \pounds DX$.

Proof: - A simple way to prove this lemma is to put a gate on line X1 and appeal to Lemma 3. Consider program LD3 Figure 11. If $\pounds X0 = \pounds X$ [*i.e.*, if the history of line X0 is initialized to $\pounds X$], then the history of



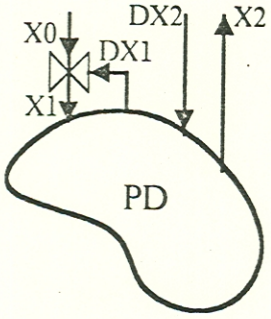
Program LD1



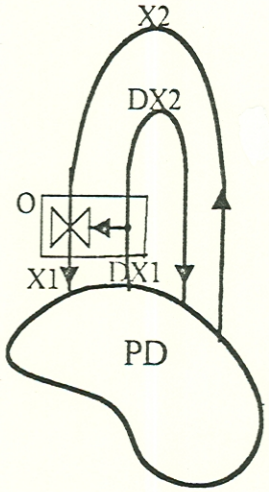
Program LD2

Figure 10: Cutting a data line and a demand line in an LD program

any line Y in program LD2 is the same as the history of the line labeled Y in program LD3.



Program LD3



Program LD4

Figure 11: Cutting a data line and a demand line in an LD program

Consider the program LD4 shown in Figure 11. The reader should verify that operator O satisfies property P2. Hence, it follows that program LD4 satisfies property P2. Therefore,

(1) $|\pounds DX1| = |\pounds DX2|$ {by definition}
(2) $|\pounds X1| = \min(|\pounds DX1|, |\pounds X2|)$
{from the semantic equation for *gate*}
$= \min(|\pounds DX2|, |\pounds X2|)$
$= |\pounds X2|$ {program LD4 satisfies property P2}

It is then easy to show that $\pounds X1 = \pounds X2 = \pounds X$ and $\pounds DX1 = \pounds DX2 = \pounds DX$. The required result now follows simply from applying lemma 3 to programs LD3 and LD4.

□

To prove P3 and P4, we will use induction on the structure of an LD program.

Proof of Property 3: (by induction on the structure of the program)

Assuming that the predicate on inputs is satisfied, we want to prove that

$$\wedge_{X \in \text{data-lines}}[|\mathcal{L}X| < |\mathcal{L}DX| \quad \Rightarrow \quad |\mathcal{L}X| = |\mathcal{H}X|]$$

The proof that P3 is invariant under juxtaposition is trivial and is omitted. To show the invariance of P3 under iteration, consider two programs F and FD which satisfy P3. Let G and GD be the programs that result from the iteration of programs F and FD respectively and let X be the line that was looped back (see Figure 12). Consider programs G and GD. If $|\mathcal{L}X| = |\mathcal{L}DX|$, then the required result follows from the inductive assumption about F and FD. Suppose $|\mathcal{L}X| < |\mathcal{L}DX|$ (we know from P2 that $|\mathcal{L}X| \leq |\mathcal{L}DX|$). If we show that in this case, $|\mathcal{L}X| = |\mathcal{H}X|$, then the required result follows from the inductive assumption about F and FD.

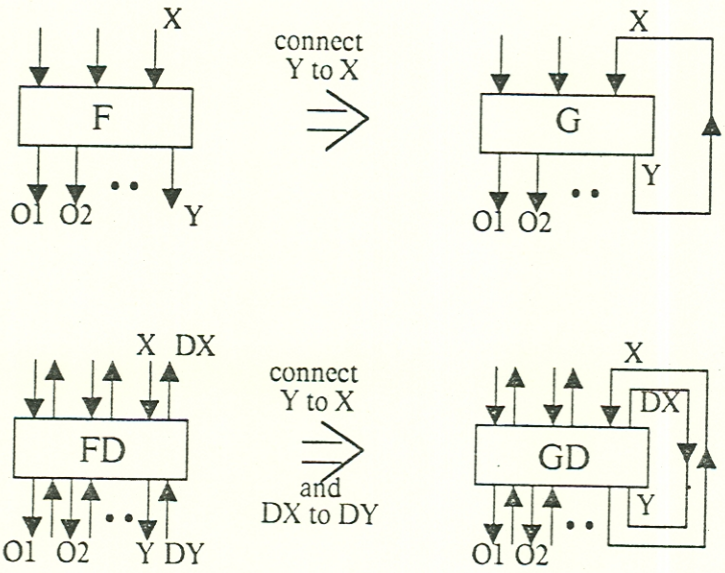


Figure 12: Iteration of two Programs

Suppose $|\mathcal{L}X| \neq |\mathcal{H}X|$. From P1, this implies that $\mathcal{L}X \subset \mathcal{H}X$. Let us denote the history $\mathcal{L}X$ by $\mathcal{A}$ and the history of line DX by $\mathcal{B}$. Consider the programs F and FD in which

$\mathcal{A}$ is fed into line X in program F,

$\mathcal{A}$ is fed in *on demand* into line X in program FD,
a demand stream of length $|\mathcal{A}|$ is fed into line DY in program FD.

Applying Lemma 4 to FD, we can conclude that the history of line Y must be $\mathcal{A}$ and the history of line LDX must be $\mathcal{B}$. From the inductive assumption about F and FD, it follows that that $\mathcal{H}$Y must be $\mathcal{A}$. By assumption, $\mathcal{H}$X is $\mathcal{A}$. We therefore have found a new fix-point for program G in which the history of all input lines is the same as before,except that the history of line Y (or X) is $\mathcal{A}$. This fix-point is distinct from the least fix-point of program G because $\mathcal{A} \subset \mathcal{H}$X. This contradicts the definition of least fix-points. Therefore, it is impossible for $\mathcal{L}$X $\subset$ $\mathcal{H}$X.

□

Proof of Property 4: (by induction on the structure of the program) Assuming the conditions on input lines are satisfied, we must show that

$$\wedge_{O \in \text{Out}} \{ |\mathcal{L}O| = |\mathcal{L}DO| \} \Rightarrow \wedge_{X \in \text{Data-lines}} \{ |\mathcal{L}X| = |\mathcal{L}DX| \}$$

The proof of this property follows the same lines as the proof of P3. As before, we only show that P4 is invariant under iteration.

Consider Figure 12 and let FD be an LD program that satisfies P4. Let GD be the resulting program when Y is connected to X and DX is connected to DY. Suppose that all output lines O*i* are satisfied - *i.e.*, $|\mathcal{L}Oi| = |\mathcal{L}DOi|$. Since program GD satisfies P2, it must be the case that $|\mathcal{L}X| \leq |\mathcal{L}DX|$. If we can show that $|\mathcal{L}X| = |\mathcal{L}DX|$, then the required result follows from the inductive assumption about program FD. We prove this by contradiction.

Suppose $|\mathcal{L}X| < |\mathcal{L}DX|$. Let us denote $\mathcal{L}$X by $\mathcal{A}$ and $\mathcal{L}$DX by $\mathcal{B}$. Since program GD satisfies P3, $\mathcal{H}$X must be $\mathcal{A}$. Consider the programs F and FD in which

$\mathcal{A}$ is fed into line X in program F,
$\mathcal{A}$ is fed in *on demand* into line X in program FD,
a demand stream of length $|\mathcal{A}|$ is fed into line DY in program FD.

By Lemma 3, it must be true that $\mathcal{H}$Y = $\mathcal{A}$. Therefore, since program FD satisfies P3, it must be true that $\mathcal{L}$Y (in program FD) must be $\mathcal{A}$. Therefore, in program FD, line Y is satisfied. By inductive assumption, program FD satisfies P4. Therefore, line X is satisfied and $|\mathcal{L}X| \leq |\mathcal{A}|$.

If $|\mathcal{L}X| = |\mathcal{A}|$, then we have found a new fix-point for program GD which is distinct from its least fix-point, since $|\mathcal{A}| < |\mathcal{B}|$. Hence, it is impossible for $|\mathcal{A}| < |\mathcal{B}|$, from which the required result follows.

Otherwise, $|\mathcal{L}X| < |\mathcal{A}|$. Let us denote $\mathcal{L}X$ by $\mathcal{A}1$. Consider the program FD in which $|\mathcal{L}DY| = |\mathcal{A}1|$, and all other inputs are the same as before. Since FD satisfies P3, all output lines O*i* of FD must still be satisfied. From P3, we can also conclude that line Y must be satisfied. Hence, line X must also be satisfied. Once again, $|\mathcal{L}X| \leq |\mathcal{A}1|$. If $|\mathcal{L}X| = |\mathcal{A}1|$, the required result follows as before. Otherwise, we can apply the same construction again. Since $|\mathcal{A}|$ is finite (since $|\mathcal{A}| < |\mathcal{B}|$), the repeated construction must terminate, at which point we will have found a fix-point for GD distinct from the least fix-point.

□

3 Conclusions and further work

In this paper, we have shown that for a powerful stream processing language L, the effect of demand-driven evaluation can be achieved by program transformation techniques. There are many dimensions along which this work can be extended.

Although the expressive power of L schemes is the same as that of recursively enumerable schemes, it is possible to add various features to L for programmer convenience. For instance, the language L did not permit any user-defined functions. If the language is extended to permit them, then the fix-point equations will contain variables of two types - sequence domains and continuous mappings between sequence domains. The data-flow graph for the language will have nodes that represent function application. An interpreter for such a language must be capable of "unfolding the function call" as soon as there is a demand for *any* of the outputs of the function. The techniques described earlier in this paper apply without modification to this embellished language.

More interesting extensions result when the semantics of streams are changed in various ways. In L, the *cons* operator is strict in its simple input. By making it non-strict in that input, it is possible to permit the elements of a stream to flow down a line out of order. More denotationally, the semantics of streams is changed to the *sub-set* ordering rather than the prefix ordering on sequences [1]. Once again, our techniques apply without any major modifications - some changes must be made for the demand propagation code for *true-gate*, *false-gate* and *merge*, but the basic ideas carry through.

An even more drastic change is to permit the elements of streams to be streams themselves. We believe that the view of a stream as a sequence of tokens flowing down a line must be abandoned in that case. However, we believe that our techniques will apply with minor modifications even in this case, but we have not investigated the problem further.

A different direction of research is to investigate other algorithms for introducing demand streams into L programs. The Microscopic Demand Propagation Algorithm given in this paper propagated demands through each operator in the dataflow graph separately, without attempting optimizations of any kind. Taking a more global approach to demand propagation would lower the overhead of introducing demands into L programs. In implementing dataflow languages with streams, it is possible to extend this idea even further and consider transformations which trade the complexity of demand propagation for a *bounded* amount of additional computation on data lines. These ideas are explored in more detail in a companion paper.

Acknowledgments: We would like to thank Gordon Plotkin, Dean Brock and Vinod Kathail for many useful discussions on streams and lazy evaluation.

References

1. Arvind, and Gostelow, K. P. Some Relationships Between Asynchronous Interpreters of a Dataflow Language. In E. J. Neuhold, Ed., *Formal Description of Programming Languages*, North-Holland, New York, 1977.

2. Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Tech. Rep. 114a, Department of Information and Computer Science, University of California, Irvine, California, December, 1978.

3. Arvind, V. Kathail, and K. Pingali. A Dataflow Architecture with Tagged Tokens. Proceedings of the 1980 International Conference on Circuits and Computers, 1980.

4. Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM 21*, 8 (August 1978), 613-641.

5. Burge, W. H. *Recursive Programming Techniques.* Addison-Wesley Publishing Co., Reading, Mass., 1975.

6. Dennis, J. B. First Version of a Data Flow Procedure Language. In *Lecture Notes in Computer Science, Volume 19: Programming Symposium: Proceedings, Colloque sur la Programmation,* B. Robinet, Ed., Springer-Verlag, 1974, pp. 362-376.

7. Friedman, D. P., and D. S. Wise. CONS Should Not Evaluate its Arguments. In *Automata, Languages, and Programming*, unknown, 1976, pp. 257-284.

8. Henderson, P. *Functional Programming: Application and Implementation.* Prentice/Hall International, Englewood Cliffs, New Jersey, 1980.

9. Jaffe, J. The Equivalence of R.E. programs and Dataflow Schemas. Tech. Rep. TM-121, Laboratory for Computer Science, MIT, Cambridge, Mass., 1979.

10. Kahn, G. The Semantics of a Simple Language for Parallel Programming. Information Processing 74: Proceeding of the IFIP Congress 74, 1974, pp. 471-475.

11. Kahn, G., and D. MacQueen. Coroutines and Networks of Parallel Processes. Information Processing 77: Proceedings of IFIP Congress 77, August, 1977, pp. 993-998.

12. Keller, R.M. and Gary Lindstrom. Applications of Feedback in Functional Programming. Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, 1981, pp. 123-130.

13. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill Publishing Company, New York, 1981.

14. D. A. Turner. The Semantic Elegance of Applicative Languages. Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, 1981, pp. 85-92.

15. Vuillemin, J. Correct and Optimal Implementations of Recursion in a Simple Programming Language. *JCSS 9* (1974), 332-352.

16. Weng, K.-S. Stream-Oriented Computation in Recursive Data Flow Schemas. Tech. Rep. TM-68, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1975.