MIT/LCS/TM-243

EFFICIENT DEMAND-DRIVEN EVALUATION (II)

Keshav Pingali

Arvind

September 1983

Efficient Demand-driven Evaluation (II)

Keshav Pingali

Arvind

19 September 1983

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge
Massachusetts 02139

Abstract

In Part I of this paper, we presented a scheme whereby a compiler could propagate demands through programs in a powerful stream language L. A data-driven evaluation of the transformed program performed exactly the same computation as a demand-driven evaluation of the original program. In this paper, we explore a different transformation which trades the complexity of demand propagation for a *bounded* amount of extra computation on some data lines.

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 (II)

1 Introduction

In Part 1 of this paper, we presented a simple but powerful stream processing language called L and described a scheme for transforming L programs whereby a data-driven evaluation of the transformed program will perform precisely the same computation as a demand-driven evaluation of the original program. The essential idea behind the scheme was to model demands for elements of a stream by associating a demand stream with that stream and permitting the compiler to introduce demand propagation operators into the dataflow program. The algorithm that the compiler used to propagate demands was the same as the one followed by a demand-driven interpreter at run-time - *i.e.*, the compiler propagated demands through each operator in the dataflow graph separately, without attempting optimizations of any kind. Let us call this the *microscopic* approach to demand propagation. For brevity, we will refer to programs resulting from such a transformation as *lazy* programs.

> **Definition 1:** A *lazy LD program* (or simply a lazy program) is one that is derived by transforming an L program using the Microscopic Demand Propagation Algorithm described in Part 1 of this paper.

Lazy programs have some very intuitively appealing properties (*i.e.*, properties P1 to P4 given in [3]), which are invariant under composition of two lazy programs. These properties formally characterize that aspect of demand-driven evaluation of programs which says that such evaluation schemes perform no useless computation, that is, all computation done by a lazy program is needed to produce the output. Thus a lazy program computes minimum histories on all data lines to produce the required histories on output lines. However, a practical notion of minimal computation should also include the overhead of demand propagation. In this paper we present a transformation which trades complexity of demand propagation for a *bounded* amount of extra computation on some data lines. Though no theoretical measure is offered to show that the new transformation presented in this paper does actually improve the over all efficiency of transformed programs, it should be clear that the new transformations will drastically reduce the

overhead of demand propagation. Without more quantitative measures, the judgment whether our technique can be characterized as an *optimization* for a class of machines (parallel or sequential) must await further theoretical analysis and post implementation evaluation of the technique.

The transformations described in this paper, like the Microscopic Demand Propagation Algorithm from Part 1, are not source-to-source transformations - rather, they transform L programs into programs in a language which is a super-set of L and which we call language LD. This language is defined in section 2 and should be already familiar to the readers of Part 1 of this paper. The set of programs that can be expressed in LD includes not only L programs and lazy programs but also programs that may be best characterized as partially demand-driven programs. In particular, the language LD allows us to define programs which are input-output equivalent of lazy programs but which do not necessarily compute minimum histories on internal lines. Since lazy programs perform the smallest amount of computation required to produce the output of the program, one way to formalize the notion of "extra computation" performed by LD programs that are *not* lazy is to define a lazy program that "corresponds" to an LD program and compare the computation performed by each program. This idea is explored in detail in section 2.

The main motivation for the work described in here arose from our research into efficient implementation of dataflow languages with streams [1]. Stream languages permit programming with infinite data objects. Therefore, a straight forward data-driven implementation of such languages will produce only partial results. Even if one is not interested in demand-driven evaluation to *minimize computation*, we need to ensure that outputs that can be computed without an infinite amount of computation are produced by the implementation. Otherwise, the implementation cannot be called correct. One way to ensure that all outputs that can be produced are produced is by introducing the notion of *fair scheduling* of function applications (*enabled computational activities* in dataflow jargon) in the machine architecture or the interpreter. For various architectural reasons, this turns out to be unsatisfactory. Therefore, data-driven implementations of stream languages can

be permitted to perform *some* additional computation over what would be performed by a demand-driven interpreter, but they must avoid performing an *infinite* amount of extra computation. We define a class of LD programs called *safe* programs for which this is true - *i.e.*, a data-driven interpreter, when executing safe programs, will perform at most a bounded amount of extra computation over what a demand-driven interpreter would. It is natural, therefore, to wonder if any L program can be transformed into a safe LD program that is more efficient than the lazy program that is generated from L. Unfortunately, the concept of safety is of limited utility since safe programs are not closed under composition, that is, the composition of two safe programs does not necessarily produce a safe program. Therefore, we will introduce the notion of *strongly-safe* programs which are that subset of safe programs that are closed under composition. It will be shown that a class of strongly-safe programs are those safe programs programs which are input-output equivalent of corresponding lazy programs. Thus, strongly-safe programs will produce the same results as lazy programs and perform, at most, a bounded amount of extra computation on data lines.

The concept of strongly-safe programs will be used in the following way. In section 3, we will first introduce a subset of L called L_0, and show how L_0 fragments of L programs may be recognized by a compiler. We then present a two step transformation that takes an L_0 fragment and produces a strongly-safe LD program that has the same input-output behavior as the lazy program corresponding to the L_0 fragment. In section 3, the first step of the transformation, which is a source-to-source transformation of L_0 (and consequently of L) programs, is described. Then, in section 4, the second step of the transformation which propagates demands (globally as opposed to microscopically) through transformed L_0 programs is given. The resulting program is shown to be strongly-safe and thus can be substituted uniformly in place of the lazy program that would have been generated from the L_0 fragment had the Microscopic Algorithm been used. In the last section, we indicate ways in which the results of this paper can be extended.

2 Language LD and Safe programs

2.1 Language LD

Language LD is a superset of language L which was described in detail in Part 1 of this paper. Language LD has four data types - *frobs*, *frob-streams*, *d* and *d-streams*.[1] Frobs encompass the usual data types like integers, reals, booleans, character strings etc. Frob-streams are sequences of frobs and are constructed by using a non-strict data constructor *cons* described below. *d* is an scalar (*i.e.*, non-stream) data type that is distinguished from all frobs. *d streams* are streams of d's. Notice that the elements of a stream must be either all frobs or all d's. As in Part 1, we will represent streams as $[a_1, a_2, ...]$ where a_1 is the scalar value that is the first element of the stream, a_2 is the scalar value that is the second element of the stream etc. The empty stream (*i.e.*, the undefined stream) is represented by [ ]. As mentioned in Part 1 of this paper, 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*. There are no difficulties in extending LD to include such a feature, but we will not do so in this paper.

The functionality of the operators of language LD 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$(b_1, [a_1, a_2, a_3, ...]) = [b_1, a_1, a_2, a_3, ...]$

Note that if the first argument of *cons* is *d*, then the second argument must be a d-stream, while if the first argument is a *frob*, then the second argument must be a frob-stream.

- *T(X, Y, ...)* - T-boxes are one-in-one-out stream operators like +, * etc. that operate "point-wise" on their inputs. Inputs to a T-box can be a combination of

---

[1] We have used the names frob-streams and d-streams instead of *data stream* and *demand stream* to avoid confusion between the names of the data types and other connotations of the words data and demand.

d-streams and frob-streams - of course, the output must be either a frob stream or a d stream. LD also has t-boxes which are like T-boxes, except that they operate on scalar values rather than streams. We will restrict t-boxes to be *total* functions.

- *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. Note that X and Y must either both be d-streams or both be frob-streams.

- *D-union(X, Y)* - X and Y must be d-streams. The output of this operator is a d-stream whose length is equal to the larger of the lengths of X and Y. The operator *d-union* is scalar version of D-union.

An LD program is a set of recursive definitions where the left hand side (LHS) of each definition consists either of a frob variable, a frob-stream variable, a d variable or a d-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 - for example, the definition of a frob stream variable cannot be the application of a D-union operator.

As before, we will find it convenient to consider an LD program as a dataflow graph. The dataflow graph corresponding to an LD 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 at the in-coming arc. Thus, there is a direct correspondence between the *history* of a line X in the dataflow graph and stream X in the LD program. In the discussion below, we will drop the distinction between the LD program and its dataflow graph, as well as the distinction between stream X in the LD program and the *history* of the line labeled X in the dataflow graph, and use these terms interchangeably.

Following Kahn [2], the semantics of LD 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. In D^{ω}, we include the empty sequence. Let V be the set containing the denotations of all frobs, and let Dem be the set containing the denotation of *d*. Consider the set $\mathfrak{I}$ containing all elements of V^{ω} and Dem^{ω}. Elements of this set can be ordered by the prefix ordering on sequences. It is easy to show that under this ordering, all the operators of LD are monotonic and continuous functions from sequences to sequences. For each equation of an LD program, we can write down a semantic equation that describes the relation between its inputs and outputs. The meaning of the LD program is the least fix point of this set of semantic equations.

2.2 Safe LD Programs

Definition 2: The *L program corresponding to an LD program* is defined to be the L program that is obtained by deleting all demand lines and all operators with demand inputs from the LD program.

Definition 3: The *lazy program corresponding to an LD program* is defined to be the LD program that is obtained by applying the Microscopic Demand Propagation Algorithm to the L program that corresponds to the LD program.

We now introduce some notation. If *FD* is an LD program, we will let *F* be the L program that corresponds to FD, and let FL be the lazy program that corresponds to FD. If X is some line in FD, we will let $\mathfrak{F}X$ stand for the "final" history of line X in program FD - (*i.e.*, the history of line X that is determined by the least fix-point of the set of equations of

---

[2] We assume unbounded buffering along each arc.

program FD) when that program is given some input. Note that if X is a data line in FD, then there will be a data line in both F and FL corresponding to it. It is convenient to let X denote these lines as well. We let $\mathfrak{F}X$ and $\pounds X$ denote the final history of line X in programs F and FL respectively.

In general, an LD program has both data lines and demand lines. In LD programs that are generated from L programs by the compiler, we may associate a demand line with a data line and assert that no token is ever produced on the data line unless there is a *d* token for it on the demand line. If X is such a data line, we will let DX denote the corresponding demand line. Let FX and FDX denote the histories of lines X and DX at any point in the computation. If *H* is any history, we let |*H*| denote the length of the history.

Definition 4: A data line X in an LD program is said to be *demand-driven by line DX* (or simply demand-driven) if and only if |FX| is always less than or equal to |FDX|.

We now want to characterize LD programs that may perform more computation than their corresponding lazy programs but do not diverge "unnecessarily" - *i.e.*, they do not diverge if the corresponding lazy programs do not diverge. Since lazy programs do not perform any unnecessary computation, this gives us a way of characterizing programs that perform a *bounded* amount of computation more than what is strictly required to produce the output of the program. We will assume that if an input data line of an LD program is demand-driven, then inputs on that line are *fed on demand* - a concept which has been explained in page 19 of Part 1 of the paper. Let FD be an LD program and let FL denote the corresponding lazy program. Let ND-In represent that subset of input data lines of FD which are not demand-driven.We define a safe program as follows:

Definition 5: A *safe* LD program diverges if and only if either there is infinite input on an input line not driven by a demand-line, or the corresponding lazy program also diverges. Formally,

$$\wedge_{I \in \text{ND-In}}[|\mathfrak{F}I| \neq \infty] \Rightarrow$$
$$(\exists X[X \in \text{Lines of FD} \wedge |\mathfrak{F}X| = \infty] \Rightarrow \exists Y[Y \in \text{Lines of FL} \wedge |\pounds Y| = \infty])$$

Note that if a user supplied infinite input on some input line that is not demand-driven,

then an infinite computation would result in a safe program. However, since inputs to the corresponding lazy program are fed on demand, it is possible that no infinite computation would result in the lazy program. If we did not have the finiteness constraint on inputs, then a large class of programs would be unsafe.

The reader can verify that any acyclic L program is safe. Similarly, any lazy program is safe. An L program that is not safe is shown below in Figure 1(ii).

(i)A Safe Program (ii)An Unsafe Program

Figure 1: A Problem with Safe Programs

Although the concept of safe programs is interesting, the major problem with it is that this property is not closed under iteration. In other words, a program that results from iterating a safe program is not necessarily safe. For example, consider the LD program consisting of a single *cons* operator shown in Figure 1(i). Since it is an acyclic LD program, it is safe. However, the program in Figure 1(ii) that results from iterating the *cons* is not. Therefore, the concept of safety is of limited use if we want to freely substitute safe programs or use them in constructing other large programs.

2.3 Input-output Equivalent Lazy Programs

It is easy to see that the main reason why safe programs are not closed under iteration is that not all input and output data lines may be demand-driven. We now want to consider those LD programs in which all input and output data lines are demand-driven.

Definition 6: An LD program FD is said to be input-output equivalent to its lazy program FL if and only if -

1. there is a one-to-one correspondence between

   a. the set of input data lines of FD and the set of input data lines of FL,

b. the set of output data lines of FD and the set of output data lines of FL,
c. the set of input demand lines of FD and the set of input demand lines of FL,
d. the set of output demand lines of FD and the set of output demand lines of FL, and

2. the histories produced on output lines of FD are the same as the histories produced on the output lines of FL when both programs are given the same inputs.

We would like to point out that for a general LD program, it may not be possible to determine whether it is input-output equivalent of its lazy program. The utility of this definition arises from the fact that LD programs are generated by the compiler from L programs - hence, this definition can act as a constraint for the compiler when it generates LD programs. We would like to note in passing that programs that are input-output equivalent to their lazy programs satisfy four properties named $P1^*$ to $P4^*$ which are very much like the properties P1 to P4 given in Part 1 of the paper. The difference between these two sets of properties is that $P1^*$ to $P4^*$ deal only with input and output lines of LD programs.

Let I be the set of input data lines and O be the set of output data lines of FD. Let IO be the union of the sets I and O.

$P1^*$ - $\wedge_{[X \in IO]} [\mathcal{F}X \subseteq \mathcal{H}X]$

$P2^*$ - $\wedge_{[X \in IO]} [|\mathcal{F}X| \leq |\mathcal{F}DX|]$

$P3^*$ - $\wedge_{[X \in IO]} [|\mathcal{F}X| < |\mathcal{F}DX| \Rightarrow |\mathcal{F}X| = |\mathcal{H}X|]$

$P4^*$ - $\wedge_{[X \in O]} [|\mathcal{F}O| = |\mathcal{F}DO|] \Rightarrow \wedge_{[Y \in I]} [|\mathcal{F}I| = |\mathcal{F}DI]$

Note that,unlike P1 to P4, $P1^*$ to $P4^*$ do not guarantee safety. This is because an LD program can be input-output equivalent to its lazy program, but there may still be an unbounded amount of computation performed on internal lines even if the corresponding lazy program performs a bounded amount of computation.

2.4 Strongly-safe Programs

Definition 7: The set of *strongly-safe* LD programs is that subset of safe LD programs that is closed under composition.

The reader may find the following "Venn diagram" useful in understanding the relationship between lazy, safe and strongly-safe programs.

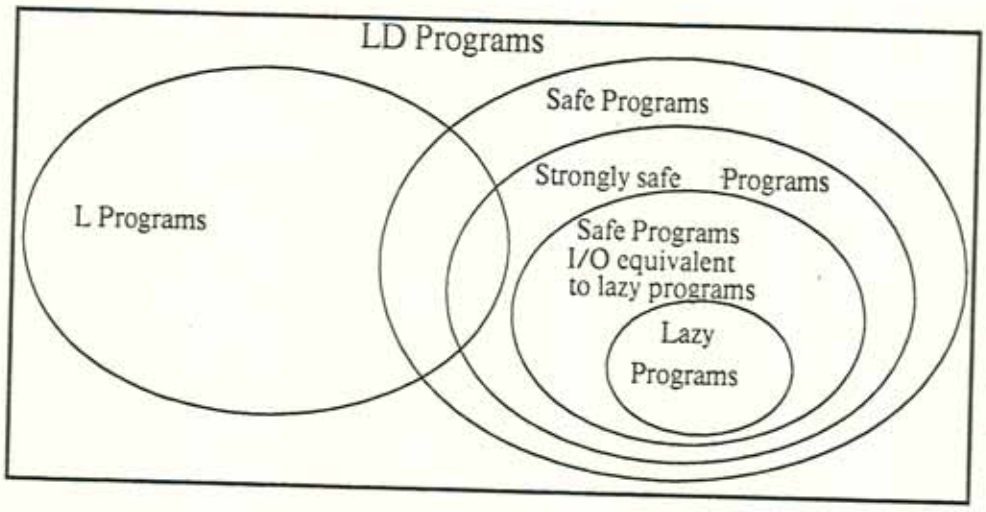


Figure 2: Lazy, Safe and Strongly-safe Programs

Theorem 8: Any safe LD program that is input-output equivalent to its corresponding lazy program is strongly-safe.

Proof: We show that the set of safe LD programs that are input-output equivalent to their corresponding lazy programs is closed under juxtaposition and iteration.

If FD and GD are two safe LD programs that are input-output equivalent to their corresponding lazy programs, it is easy to see that the juxtaposition of FD and GD is a safe LD program that is also input-output equivalent to its corresponding lazy program.

We now show that the set is closed under iteration. Let HD be the program that results from the iteration of some line X in program FD, as shown in Figure 3. Let FL be the lazy program that corresponds to FD and let HL be the lazy program that corresponds to HD. Consider programs HD and HL when both are given the same inputs. We first show that the histories of lines X and DX in both programs HD and HL must be the same. It is easy to see that the histories of

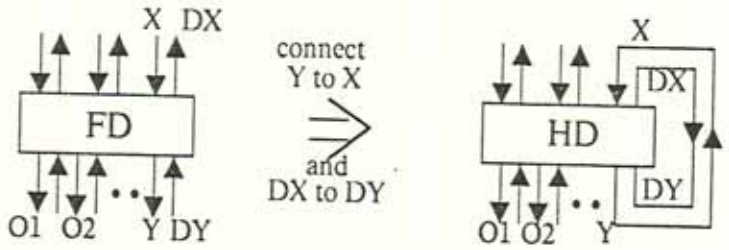


Figure 3: Iteration of a Safe Program

input and output lines of HD at any fix-point of HD must be the histories of the input and output lines of HL at a fix-point of HL and vice versa. Therefore, it follows that the histories of lines X and DX in both programs must be the same. Since the functionality of FD and FL are the same, it now follows that HD is input-output equivalent to HL. The safety of HD now follows from the the inductive assumption about the safety of FD and the fact that the histories of X and DX are the same in HD and HL.

□

3 A Source-to-Source Transformation of L programs

In this section, we first define a subset of L called L_0 and show how L_0 fragments of L programs can be identified. We show that the compiler can take an L_0 fragment and transform it into a safe program that is input-output equivalent to its corresponding lazy program. This transformation is done in two steps - we present in this section the first step which is a source-to-source transformation that converts any L_0 fragment into a canonical form which has a simple loop (iterative) structure.

3.1 The Language L_0

An L_0 program is a deadlock-free L program in which no *merge*, *true-gate* or *false-gate* operators are present. The deadlock-free property (often referred to as *liveness* in Petri net literature) can be tested by some kind of a *cycle-sum* test such as that of Wadge [4]. For example, we can assign a 0 to every T-box, 1 to every *cons*, and -1 to every *first* and *rest*. For an L_0 program to be deadlock-free, the sum of the integers in every cyclic interconnection of operators must be greater than zero.

We will also assume that L_0 programs have been converted into a canonical form in which

- there is no *cons* box whose output is directly connected to a *rest* or a *first* box
- every circular interconnection of boxes has at least one T-box in it.
- every output is the output of a T-box.

L_0 programs that do not meet these criteria can be converted into the canonical form by repeated use of the following transformations -

- any occurrence of *first(cons(a, B))* is replaced by a
- any occurrence of *rest(cons(a, B))* is replaced by B[3]
- one identity T-box is introduced into every cycle that does not have any T-box. The scalar function associated with the identity T-box is the identity function on atomic values
- introduce one identity T-box at each output that is not the output of a T-box.

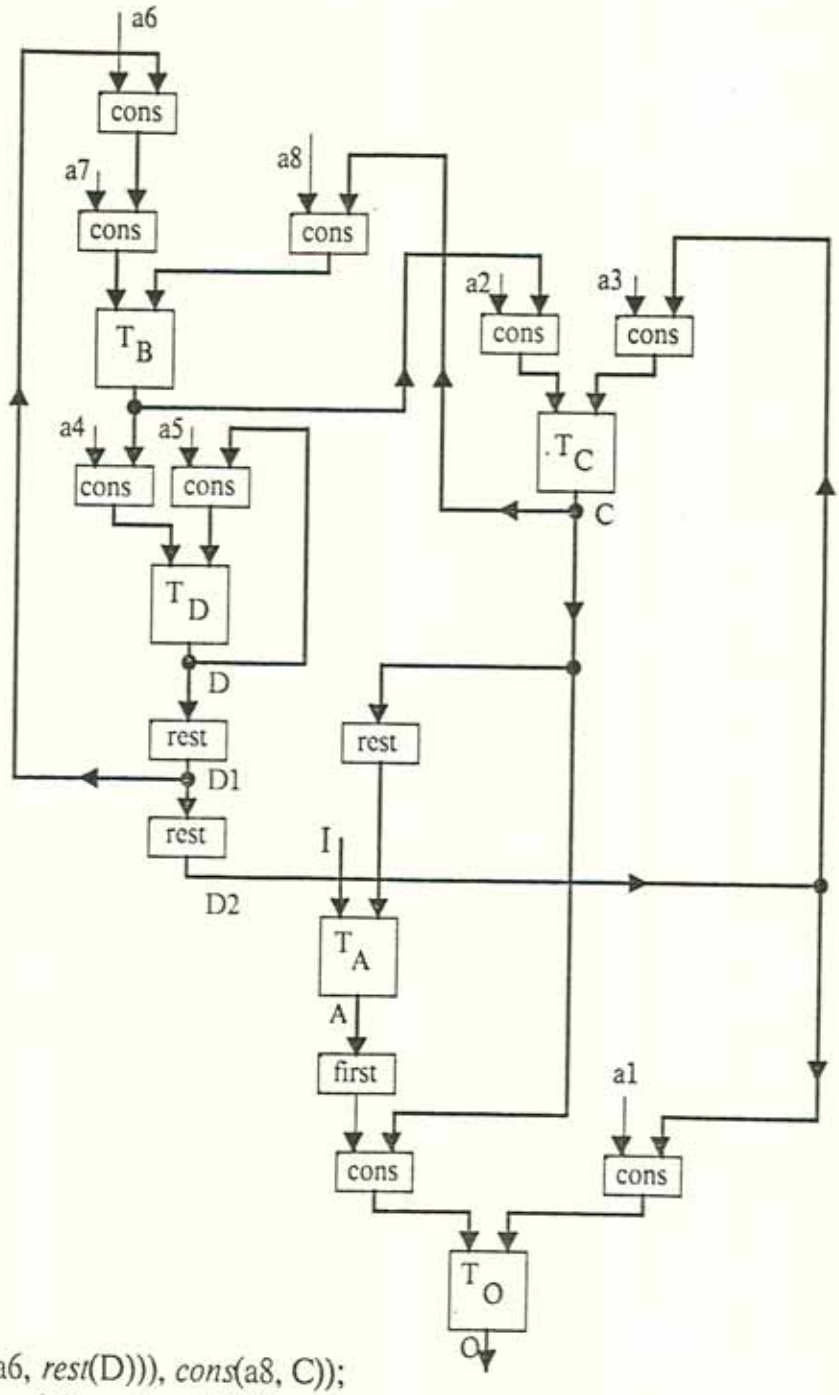
Given an L program, we can identify all L_0 fragments in it by deleting all *true-gates*, *false-gates* and *merge* operators from the program. Each connected program graph that remains is an L_0 fragment.

3.2 Transforming L Programs into Strongly-Safe Programs

Figure 4 is an example of an L_0 program. Consider the elements of stream O in the program of Figure 4 (This example program is contrived and fairly complex. However it is useful for illustrating many different aspects of our technique, and will be used throughout this section). The basic source of complexity (or lack of structure) in an L_0 program has to do with the computation of the first few elements of the output stream. In the program of Figure 4, the first element depends upon A(1) and a1, while the second element depends upon C(1) and D(3) (it is easier to make these observations in the graph version). After the second element the pattern is obvious; the k^{th} element of O will depend upon C(k-1) and D(k+1). Intuitively we can say that the program reaches a "steady-state" after computing the first few elements of the output stream. We will show that every L_0 program can be

---

[3] This transformation is not valid if there is a possibility that *a* may be undefined.



$A = T_A(I, rest(C));$
$B = T_B(cons(a7, cons(a6, rest(D))), cons(a8, C));$
$C = T_C(cons(a2, B), cons(a3, rest(rest(D))));$
$D = T_D(a4, cons(a5, B));$
$O = T_O(cons(first(A), C), cons(a1, rest(rest(D))));$

Figure 4: An L_0 Program

transformed into an L_0 program which consists of three acyclic graphs called *adjustments*, *prelude* and *steady-state* connected according to the schema shown in Figure 5. In the *prelude*, only those elements of the output stream which do not conform to the general computing pattern are computed. The *adjustments* is an acyclic interconnection of *first* and *rest* operators, and is used to select those elements of input streams which are needed by the *prelude* and the *steady-state* parts. The steady-state acyclic graph has the one-in-one-out property - that is, one set of input values will produce one output value on each line. After giving a definition of *steady-state*, we will prove that every L_0 program does reach a *steady-state* after computing a finite number of output elements and give an algorithm for transforming any L_0 program into the schema of Figure 5.

Once L_0 programs have been transformed into the schema of Figure 5, it is easy to introduce gates and demand propagation code into the program in order to make it strongly-safe. By introducing gates into the steady-state graph as shown in Figure 6, we can ensure that an iteration of the steady-state graph will be executed only when there is a demand for some output. In this way, the transformed program is made safe. We then propagate demands for the outputs to demands for the inputs, thereby ensuring that the input-output behavior of the transformed program is the same as that of the corresponding lazy program. This demand propagation is done by a *global* algorithm (as opposed to a microscopic algorithm like Algorithm-MDP of Part 1). By Theorem 8, the program that results from these transformations is strongly-safe.

We will now describe how the transformation for L_0 programs that was outlined above can be used to make L programs strongly-safe. The algorithm given below essentially identifies L_0 fragments of L programs and transforms them into programs that correspond to the strongly-safe schema of Figure 6. Demand propagation through *true-gates*, *false-gates* and *merges* is done as specified by Algorithm-MDP. Since the composition of strongly-safe programs is another strongly-safe program, the result of the transformation is a strongly-safe LD program.

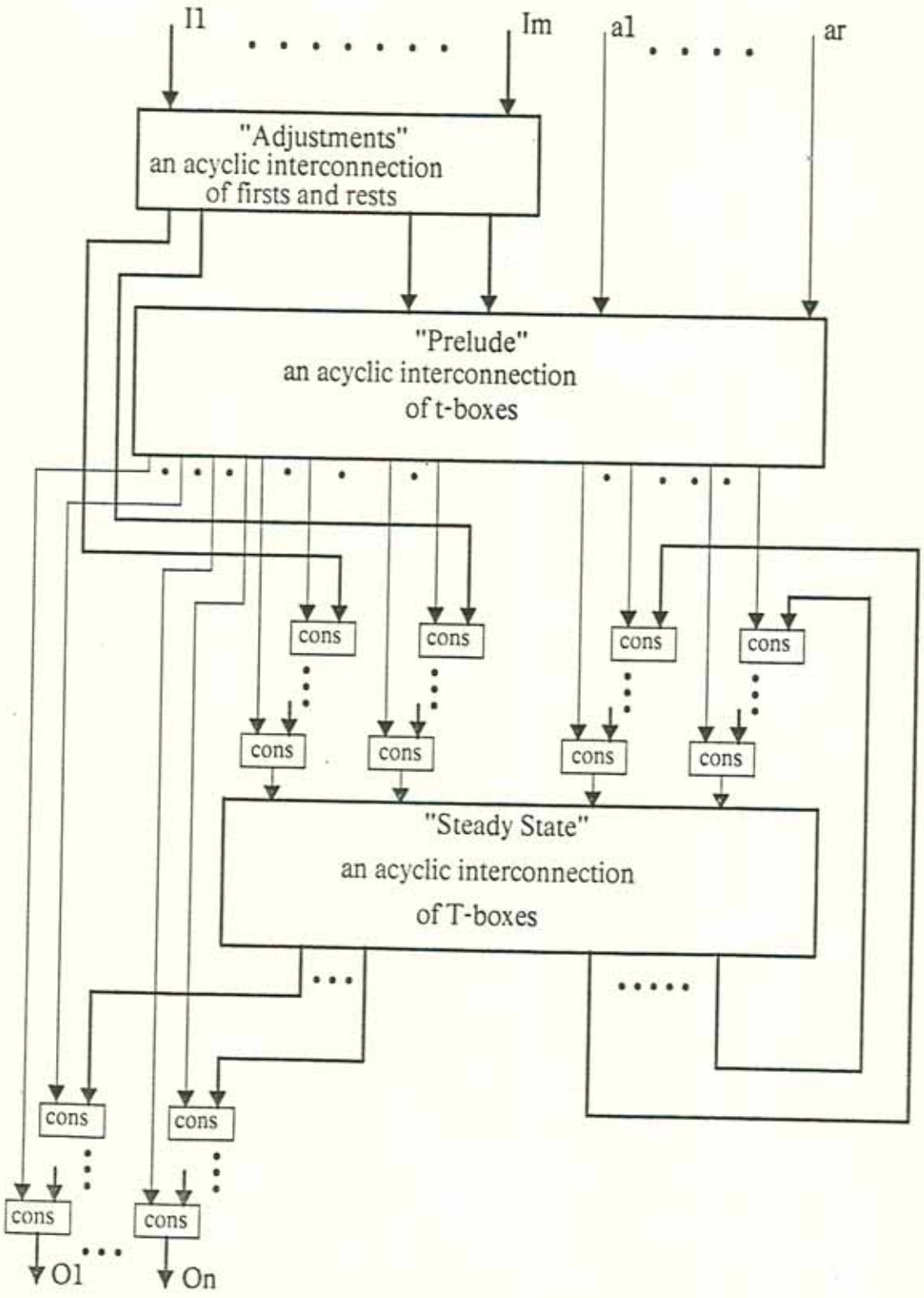


Figure 5: Schema for Canonical (simple loop) Programs in L_0

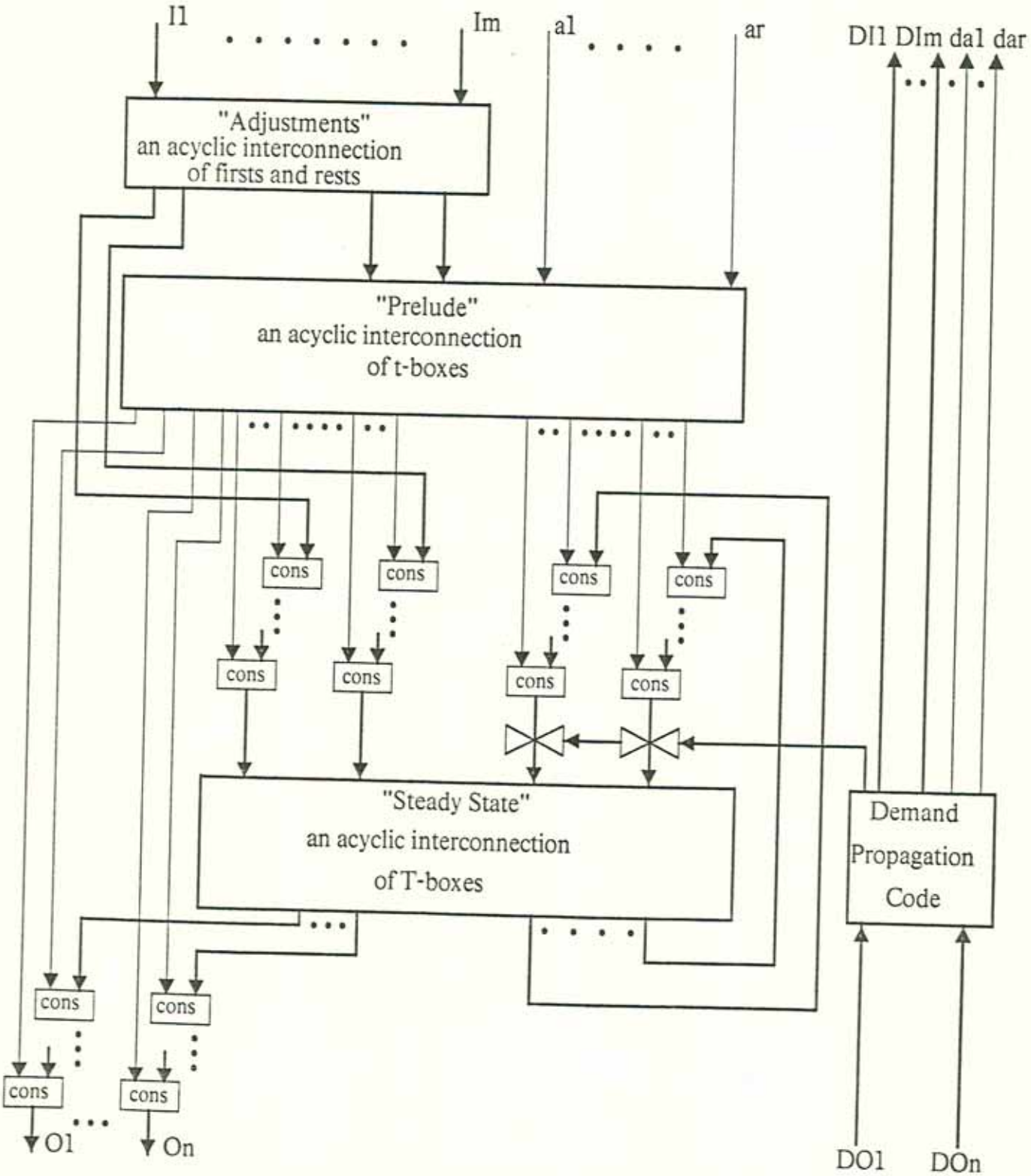


Figure 6: Schema for Strongly-Safe Programs in L_0

Algorithm-TLSS : An algorithm for transforming L programs into strongly-safe LD programs.

1. Delete all true-gates, false-gates and merges from the L program and declare all inputs to these operators as outputs of the remaining program. Each connected graph in the resulting graph is an L_0 program.

2. Transform each L_0 fragment into the schema of Figure 5 by using Algorithm-SST (see section).

3. Replace each L_0 component in the original graph with the corresponding transformed component. Since a transformed component has exactly the same inputs and outputs as the untransformed component, this operation is well-defined.

4. Introduce demand propagation code and gates into each transformed L_0 fragment by using Algorithm-IOE and Algorithm-GDP (see section) thereby producing programs that correspond to Figure 6. Introduce demand propagation code for true-gates, false-gates and merges using Algorithm-MDP.

□

Figure 7 illustrates this algorithm for the case when the L program has a *true-gate* and a *merge* operator, and one L_0 fragment.

3.3 Dependency Matrices and Predecessor Paths

The algorithm for transforming L_0 programs into the schema of Figure 5 examines the data dependencies of elements of output streams. A convenient data structure to record the data dependencies of elements of streams in a program is a *data dependency* matrix that has a row for every t-box and T-box in the program. If stream X is the output of a T-box T_X, we will record the data dependencies of stream element X(i) in the i^{th} column of the row for X. A row for a t-box will have entries only in its first column since the output of a t-box is an scalar value. Depending on the context, we will use X(n) to mean either the n^{th} element of stream X (as was done above) or the n^{th} column of the row for stream X in the dependency matrix. If an L_0 program has only *one* output stream, we will name that stream O. If the program has more than one output stream, we will assume that the streams are named O1, O2, ... We will refer to the set of all output streams as $\{Oi\}$.

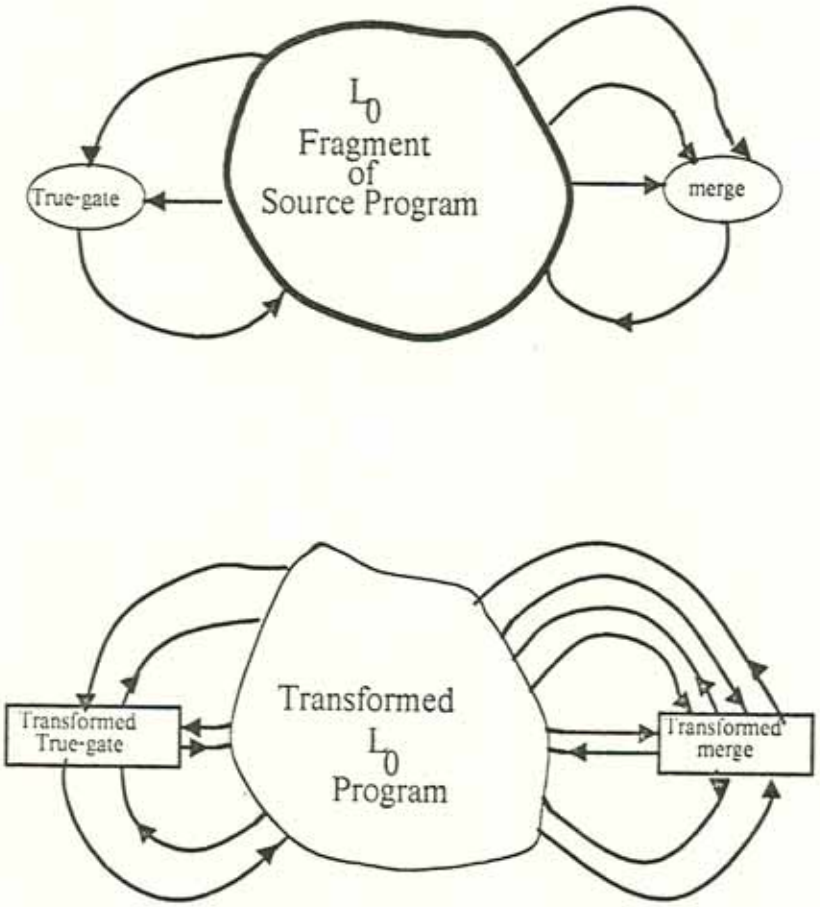


Figure 7: Transformation of L Programs

We now introduce the notion of a *predecessor path* in order to make entries in the data dependency matrix.

Definition 9: A *predecessor path* in an L_0 program is a path obtained by tracing backwards (*i.e.*, in a direction opposite to the flow of data) in the graph from the input of a t-box or a T-box with the following rules :

1. When the output of a *first* or a *rest* is encountered, the tracing is resumed from the input of the operator. We shall say that the *first* or the *rest* lies on the predecessor path.

2. When the output of a *cons* is encountered, one of the inputs to the *cons*, is arbitrarily chosen and the tracing is resumed. As before, we shall say that the *cons* lies on the predecessor path.

3. The trace terminates when the output of a t-box or a T-box, a stream input or an scalar input is encountered.

Lemma 10: For programs in L_0, all traces terminate.

Proof: Straight-forward, from the fact that every cycle has a T-box in it.

□

Notice that a finite ordered sequence of arcs is a *predecessor path* if and only if the path connects the input of a t-box or the input of a T-box to either the output of a t-box, the output of a T-box, or an scalar input or a stream input, and intermediate operators in the path are only *cons*'s, *first*'s and *rest*'s.

Lemma 11: Any predecessor path in a graph must be of one of the five types shown in Figure 8.

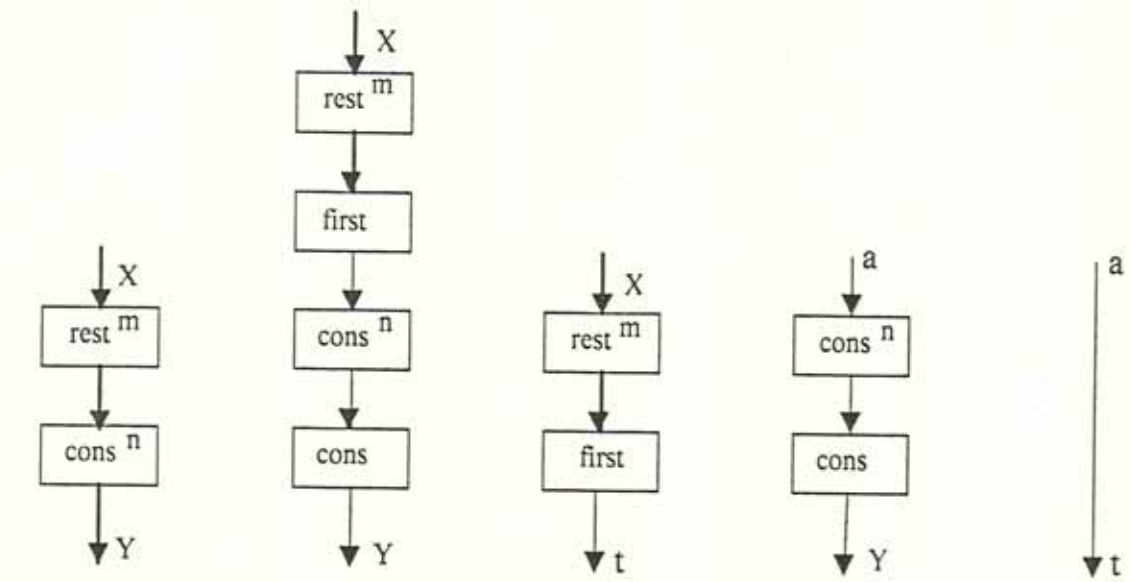
Proof: The proof is a straight-forward induction on the length of a predecessor path. A predecessor path of length n has one more *first*, *rest* or *cons* on it than a predecessor path of length (n-1). Assume that all predecessor paths of length n-1 are of one of the five types shown in Figure 8. It is easy to show that inserting another *first*,*rest* or *cons* anywhere on a path of length n-1 would result in either an illegal interconnection or in a predecessor path of length n that is of one of the five types shown in Figure 8.

□

The dependency matrix of a program contains an entry for every predecessor path in the program. The entries are made according to the rules given in Figure 8. The dependency matrix for the program in Figure 4 is shown in Figure 9. Note that the starred entries are a short-hand to represent an infinite number of entries in a row. For example, if $X(n)^*$ occurs in the Y(k) position, it means that Y(k) depends on X(n), Y(k+1) depends upon X(n+1) etc. Furthermore, by replacing $X(n)^*$ in the Y(k) position with X(n) and making an additional entry $X(n+1)^*$ in the Y(k+1) position, (an operation which from now on will be referred as *unrolling*) we get the dependency matrix of an equivalent L_0 program.

Definition 12: If $Y(j)^*$ occurs in the i^{th} column of row X in the dependency matrix, then the dependencies of X(i) on Y(j), X(i+1) on Y(j+1), ... are said to be *starred dependencies.*

For example, the dependency of B(3) on C(2) is a starred dependency while the



X is a stream input or the output of a T-box

Y is an input of a T-box

a is a scalar input or the output of a t-box

t is the scalar input of a *cons* or an input of a t-box

Figure 8: Predecessor Paths and the Corresponding Matrix Entries

dependency of O(1) on A(1) in *not* a starred dependency. For the first argument to T-box T_O, the starred dependencies are O(2) on C(1), O(3) on C(2) .... The reader can test his understanding of the dependency matrix by checking that O(3) depends on { a_2, a_3, a_4, a_5, a_6, a_7, a_8, B(1), B(2), B(3), C(1), C(2), D(1), D(2), D(3) and D(4)}.

Lemma 13: For any stream X in an L_0 program and any integer k, the number of elements in the transitive closure of the data dependencies of X(k) (written as $\mathfrak{T}(X(k))$) is finite[4].

Proof: Construct a tree that corresponds to the transitive closure of the data

[4]Note that X(k) is not a member of $\mathfrak{T}(X(k))$

	1	2	3
A	I(1)*,C(2)*		
B	a7,a8	a6,C(1)*	D(2)*
C	a2,a3	B(1)*,D(3)*	
D	a4,a5	B(1)*, D(1)*	
O	A(1),a1	C(1)*,D(3)*	

Figure 9: The Dependency Matrix for the Program in Figure 4

dependencies of X(k). By Konig's lemma, the number of nodes in a tree that has finite out-degree at every node is infinite if and only if there exists an infinite branch in the tree. Consequently, if the number of elements in $\mathcal{T}(X(k))$ is infinite, there must be an infinite branch in the tree. Each node in the tree is labeled by a stream element. Since the number of T-boxes in the program is finite, there must be at least one T-box whose elements occur infinitely often in the infinite branch. Let that T-box be Y. It must be the case then, that some element Y(j) must depend on at least one element of the form Y(n) where $n \geq j$. This can happen only if there is some cycle in the program that fails the cycle-sum test. Since such cycles are ruled out in L_0, it follows that the number of elements in $\mathcal{T}(X(k))$ for any X(k) must be finite.

□

3.4 Steady-State of L_0 Programs

Consider an L_0 program with one output stream O. In order to compute O(k) (for any $k \geq 1$), it is necessary to compute all the elements in the transitive closure of the data-dependencies of O(k) - *i.e.*, all the elements of $\mathcal{T}(O(k))$. However, since our semantics for streams dictates that O(1), ... O(k-1) must have been computed before O(k) can be computed, the only "new" elements that must be computed are those in $\mathcal{T}(O(k))$ and which were not required for the computation of O(1), O(2), ... , O(k-1). If a program fragment corresponding to the source program can be identified whose repeated evaluation will produce "new-elements" at k_0+1, k_0+2, ... for some integer k_0 then the source program can be said to be in a "steady-state" after k_0. We formalize this concept of steady-state by

first defining the set *new-elements*(O,k) as shown below. In L_0 programs with more than one output stream, we will be interested in the set of "new" elements required to compute all elements O1(k), O2(k) ..., assuming that O1(1), ..., O1(k-1), O2(1), ..., O2(k-1), ... have all been computed.

For any output stream Oi, the set *new-elements*(Oi,k) is defined as follows:

new-elements(Oi,1) = $\mathcal{T}$(Oi(1)) U {Oi(1)}
new-elements(Oi,k) = $\mathcal{T}$(Oi(k)) U {Oi(k)} −
{new-elements(Oi,1) U..U new-elements(Oi,k-1)}.

where "U" and "−" represent the set union and difference operations respectively.

We define the set *new-elements*(k) as follows:

new-elements(1) = new-elements(O1,1) U new-elements(O2,1) ...
new-elements(k) = new-elements(O1,k) U new-elements(O2,k) ...
−{new-elements(1) U ... U new-elements(k-1)}

It should be clear that for L_0 programs with only one output, the set new-elements(k) is equal to the set new-elements(O,k).

The sets new-elements(1), ..., new-elements(5) for the program in Figure 4 are shown below :

new-elements(1) = {a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,
I(1),A(1),B(1),B(2),C(1),C(2),D(1),D(2),D(3),O(1)}
new-elements(2) = {O(2)}
new-elements(3) = {B(3),D(4),O(3)}
new-elements(4) = {B(4),C(3),D(5),O(4)}
new-elements(5) = {B(5),C(4),D(6),O(5)}

The program for computing new-elements(1), ..., new-elements(5) can be derived by following data dependencies in the dependency matrix, and is shown below -

$D(1) = t_D(a4,a5);$
$C(1) = t_C(a2,a3);$
$B(1) = t_B(a7,a8);$
$D(2) = t_D(B(1),D(1));$
$B(2) = t_B(a6,C(1));$
$D(3) = t_D(B(2),D(2));$
$C(2) = t_C(B(1),D(3));$
$A(1) = t_A(I(1),C(2));$
$O(1) = t_O(A(1),a1);$! end of program to compute O(1) !
$O(2) = t_O(C(1),D(3));$! end of program to compute O(2) !
$B(3) = t_B(D(2),C(2));$
$D(4) = t_D(B(3),D(3));$
$O(3) = t_O(C(2),D(4));$! end of program to compute O(3) !
$C(3) = t_C(B(2),D(4));$
$B(4) = t_B(D(3),C(3));$
$D(5) = t_D(B(4),D(4));$
$O(4) = t_O(C(3),D(5));$! end of program to compute O(4) !
$C(4) = t_C(B(3),D(5));$
$B(5) = t_B(D(4),C(4));$
$D(6) = t_D(B(5),D(5));$
$O(5) = t_O(C(4),D(6));$! end of program to compute O(5) !

The pattern after $k=4$ appears to be fixed, that is, to compute O(k) we need to compute only B(k), C(k-1), and D(k+1). But interestingly enough if we are willing to compute a few elements more than once, the computation of O(2) and O(3) can be fitted in the same pattern (O(3) requires recomputation of C(2) while O(2) requires recomputation of B(2), C(1) and D(3)). As we shall show later, recomputation of a few elements may actually reduce the size of the transformed program because there are fewer special cases to be dealt with in the *prelude* part.

We now define *steady-state.*

Definition 14: A program is said to have reached *steady-state* at k_0 if

1. no scalar input or output of a t-box is in new-elements(k) for $k \geq k_0$, and

2. for all $i \geq 0$, $X(k_X)$ belongs to new-elements(k_0) if and only if $X(k_X+i)$ belongs to new-elements(k_0+i).

Definition 14 is not an algorithmic definition of steady-state since it involves the computation of new-elements(j) for all j. We now give an operational definition of steady-state which will enable us to compute the value of k at which an L_0 program reaches steady-state. Our new definition of steady-state involves checking that new-elements(k) satisfies three conditions.

Condition 1: All dependencies between members of new-elements(k) are starred dependencies.

Condition 2: If $X(k_X)$ is a member of new-elements(k), then $X(k_X+1)$, $X(k_X+2)$, ... do not belong to new-elements(1) U ... U new-elements(k).

Before presenting the third condition, which is rather complex, we motivate its need. Lemma 16 shows what Conditions 1 and 2 guarantee. The proof of Lemma 16 requires the following property of the set new-elements(k).

Lemma 15: If $X(k_X)$ is in new-elements(k) and is not of the form Oi(k), then there must be some Oi(k) in new-elements(k) such that in the dependence graph of Oi(k), there is a path from Oi(k) to $X(k_X)$ in which all intermediate nodes (if any) are in new-elements(k).

Proof: From the definition of new-elements(k), $X(k_X)$ must be in at least one set new-elements(Oi, k), which implies that it is in $\mathcal{T}$(Oi(k)). Since $X(k_X)$ is in new-elements(k), Oi(k) and any intermediate nodes in the path from Oi(k) to $X(k_X)$ in the dependence graph of Oi(k) must be in new-elements(k) - otherwise, some node Y(j) in this path, and all elements in $\mathcal{T}$(Y(j)) (which includes $X(k_X)$) will be in new-elements(1) U ... U new-elements(k-1). This will contradict the definition of new-elements(k).

□

Lemma 16: If Conditions 1 and 2 are true at some k and $X(k_X)$ belongs to new-elements(k),then $X(k_X+1)$ belongs to new-elements(k+1).

Proof: From Lemma 15, there must be some O(k) in new-elements(k) such that $X(k_X)$ is in $\mathcal{T}$(O(k)) and all intermediate nodes (if any) on the path from O(k) to $X(k_X)$ in the dependence graph of O(k) are in new-elements(k). By Condition 1, all dependencies between members of new-elements(k) are starred - hence, $X(k_X+1)$ is in $\mathcal{T}$(O(k+1)). Condition 2 now guarantees that $X(k_X+1)$ is in new-elements(k+1).

□

Unfortunately, Conditions 1 and 2 are *not* strong enough to guarantee that these will be

the only elements in new-elements(k+1). The point is best illustrated by means of the following example:

$O = T_O(cons(a,cons(a,cons(b,I))),\ I)$

The reader can verify that new-elements(2) = {O(2),I(2)} and that Condition 1 and Condition 2 are satisfied at k = 2. As expected, O(3) and I(3) are members of new-elements(3) - however, so is b. To rule out such occurrences in the steady-state, we need to look for patterns in the inputs required to compute new-elements(k). We will use the following definition to define the required input elements for computing new-elements(k).

> **Definition 17:** The set *minimal-inputs*(1) is defined to be the empty set. For k > 1, *minimal-inputs*(k) is defined to be the union of the subset of new-elements(1) U ... U new-elements(k-1) whose members are in in the immediate data dependencies of elements of new-elements(k), together with the set of all elements Oi(k) that are not in new-elements(k).

This definition deserves further explanation. For one output L_0 programs, minimal-inputs(k) is simply the set of previously computed elements which are in the immediate data dependencies of elements in new-elements(k). The sets minimal-inputs(1), ..., minimal-inputs(5) for the program whose dependency matrix is shown in Figure 9 are given below -

minimal-inputs(1) = {}
minimal-inputs(2) = {C(1), D(3)}
minimal-inputs(3) = {C(2), D(2), D(3)}
minimal-inputs(4) = {B(2), D(3), D(4)}
minimal-inputs(5) = {B(3), D(4), D(5)}

For multiple output L_0 programs, the situation is a little more complicated. As in the one output case, minimal-inputs(k) contains all previously computed elements which are in the immediate data dependencies of elements in new-elements(k). In addition, it will contain those elements of {O*i*(k)} that are not in new-elements(k). Consider the dependency matrix of Figure 9. Suppose both O and D were output streams. Since D(2) has been computed by the first stage, it is not in new-elements(2). Our definition of minimal-inputs requires, then, that D(2) be an element of minimal-inputs(2). This distinction may seem very minute but its importance will become evident when we generate code for the transformed L_0 program.

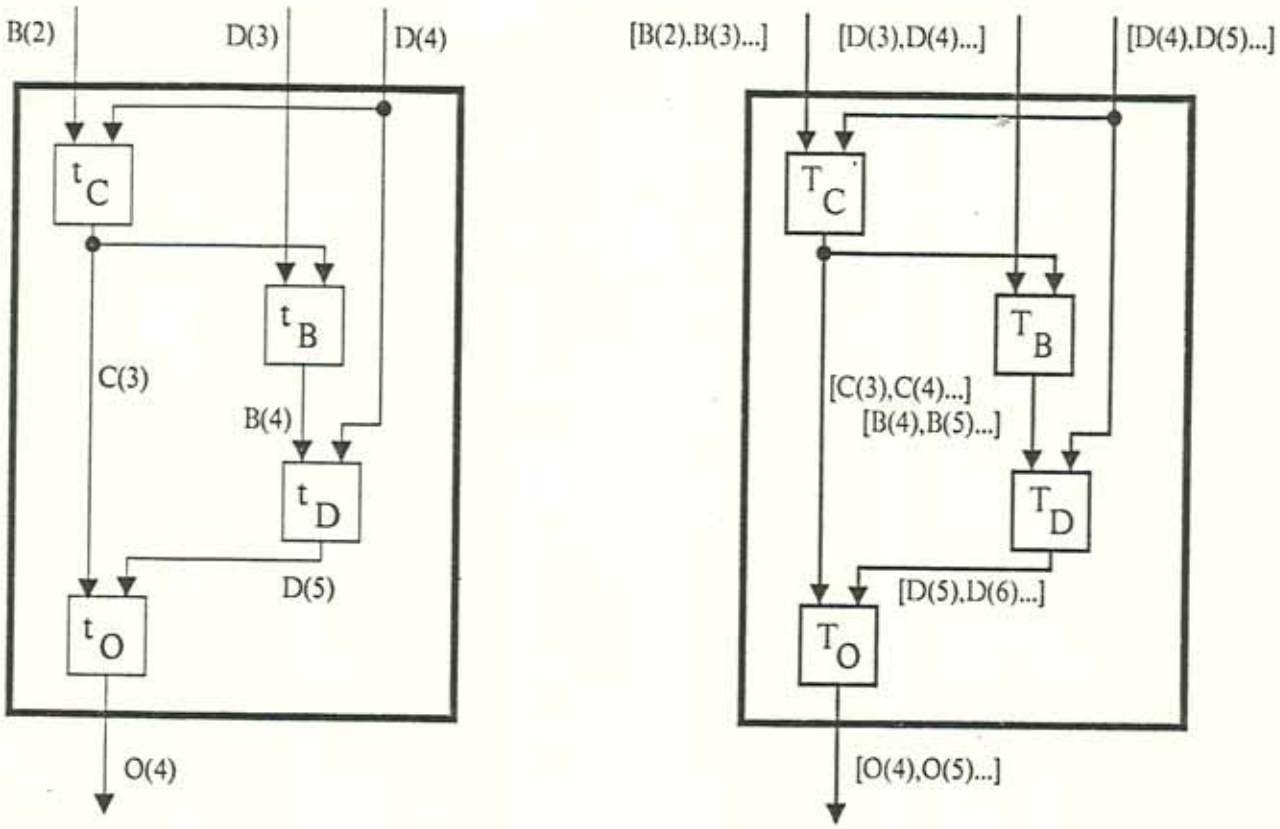
For the example we have been considering, the pattern of minimal inputs does not seem to change after $k \geq 4$. We ask the reader to take it on faith that the pattern of new elements and minimal inputs needed to compute O(k) does not change after $k=4$ in the example under consideration. Later in this section, we will prove that this is indeed the case. The reader is warned that in general, a program can reach steady-state at some k_0, even if the pattern of minimal-inputs is not fixed for $k \geq k_0$. In what follows, we will actually motivate and define a weaker but unfortunately, more complicated condition to detect steady-state. This weaker condition, known as Condition 3, also yields $k = 4$ as the value of k at which the program reaches steady-state - this example was chosen so that we could make a point about steady-state without giving the complicated condition straight-away. Condition 3 can be motivated best by considering the code to be generated for the steady-state part.

If the pattern of inputs indeed, does not change after $k=4$ then in order to compute O(k) for $k \geq 4$, we will need to compute B(k), C(k-1) and D(k+1), and get elements B(k-2), D(k-1), and D(k) from previous stages as shown below.

$C(k-1) = t_C(B(k-2),D(k));$
$B(k) = t_B(C(k-1),D(k-1));$
$D(k+1) = t_D(B(k),D(k));$
$O(k) = t_O(C(k-1),D(k+1));$

An L_0 program that computes O(4) is shown in Figure 10(i). An L_0 program that computes O(k) for $k > 3$ is shown in Figure 10(ii). The reader can verify that if [B(2),B(3),...], [D(4),D(5),...] and [D(3),D(4),...] are input to this graph it will produce [O(4),O(5),...].

How should the input streams for the program in Figure 10(ii) be generated ? Notice that, in Figure 10(ii) the output of the T_B-box is [B(4),B(5),...] while the output of the T_D-box is [D(5),D(6),...]. By providing "feed-back" paths in the graph, we can take elements of the B and D streams generated during the computation of O(k), and feed them back into the inputs to be used for computation of more elements of O. A program to accomplish this is shown in Figure 11. Hence, given B(2), B(3), D(3) and D(4), the graph of Figure 11 will produce [O(4), O(5), ...]. The next question is, can the input elements for the steady-state part be identified easily. It is easy to see that minimal-inputs for the stage



(i) Program for computing O(4) (ii) Program for computing [O(4), O(5), ...]

Figure 10: The Acyclic Steady-State Graph for the Program of Figure 4

when the program enters steady-state (k=4, for this example) must be needed for the steady-state part of the graph. To understand why inputs other than the minimal-inputs are needed, consider B(3) which does not belong to minimal-inputs(4). B(3) is *not* required for computing O(4) but is needed to compute O(5)! Its value must be generated in the prelude since the T_B-box in the Figure 11 generates only B(4),B(5),.... .

We require that the value of B(3) be available even during the computation of O(4) for the reasons of simplifying code generation. If this requirement was not made, then even after the program reaches steady-state, the computation of elements of the output stream will involve computation in both the steady-state and the prelude parts of the graph -- not a desirable situation for code generation. Elements such as B(3), as shown below, are precisely the elements of the form X(m) where X(i) belongs to minimal-inputs(k), X(n) belongs to new-elements(k) and $i < m < n$.

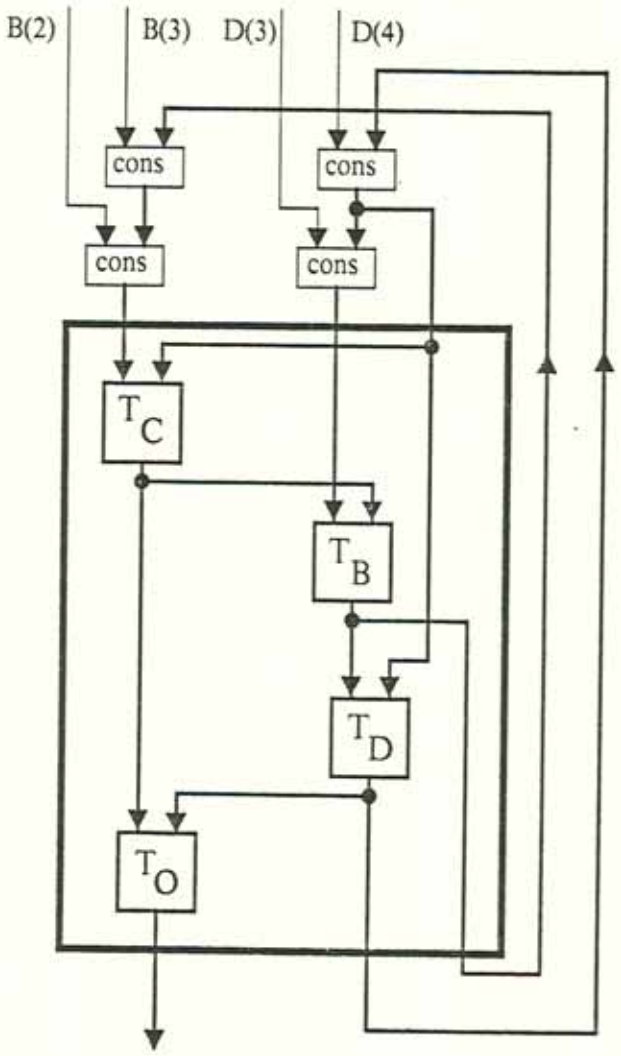


Figure 11: The Steady-State part of the Transformed Program of Figure 4

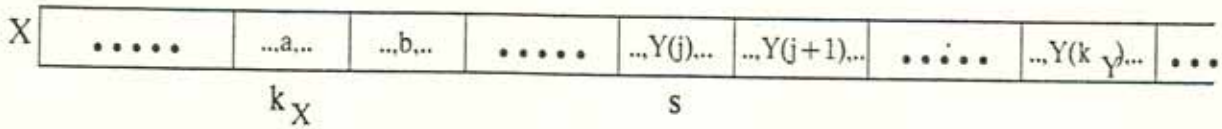


Figure 12: A Portion of a Dependency Matrix

The portion of a dependency matrix shown in Figure 12 can help further in motivating Condition 3. Let X and Y be two T-boxes, and let $Y(j)^*$ be the starred entry in X(s). This entry has been unrolled in Figure 12. Let $X(k_X)$ and $Y(k_Y)$ be in new-elements(k) at some

k at which Conditions 1 and 2 are satisfied. We would like to assert that if the program reaches steady-state at k, then $X(k_X)$, $X(k_X+1)$ ... will be computed in the steady-state program. From the dependency matrix of Figure 12, this means that [a, b, ..., Y(j), Y(j+1), ..., $Y(k_Y)$, $Y(k_Y+1)$, ...] must be input into T_X, the T-box that computes [$X(k_X)$, $X(k_X+1)$ ...] in the steady-state program. Now, the stream [$Y(k_Y)$, $Y(k_Y+1)$, ...] is generated in the steady-state code. Consequently, the values of a, b,..., Y(j), Y(j+1),...,$Y(k_Y-1)$ must be generated in the prelude. Once the program reaches steady-state, we do not want computation to take place in both the prelude and the steady-state program - hence, we require that a,b,..., Y(j), Y(j+1),..., $Y(k_Y-1)$ must be computed before the program reaches steady-state. As we have seen in a small example earlier Conditions 1 and 2 are not adequate to ensure this. We therefore impose Condition 3, which we will check only if new-elements(k) satisfies Conditions 1 and 2.

Condition 3: For each element α in minimal-inputs(k), perform the following checks -

1. if α is an element of the form Oi(k), check that some j > k, element Oi(j) is in new-elements(k), and that all elements Oi(k+1), Oi(k+2), ..., Oi(j-1) are in new-elements(1) U ... U new-elements(k-1).

2. for each element $X(k_X)$ in new-elements(k) that depends directly on α, perform the following check -

   a. if this dependency is starred, then α is of the form Y(i) where Y is some stream input or the output of a T-box. Check that for some j > i, Y(j) is in new-elements(k), and that Y(i+1), ..., Y(j-1) are in new-elements(1) U ... U new-elements(k-1).

   b. If the dependency is not starred, let α be the p^{th} argument to the function that computes $X(k_X)$ and let s be the smallest integer > k_X such that the p^{th} argument of X(s) is starred and is, say, Y(j). Check that for some $k_Y \geq$ j, $Y(k_Y)$ is in new-elements(k), and that the elements in the immediate data dependencies of $X(k_X+1)$, ..., X(s-1), as well as Y(j),Y(j+1), ..., $Y(k_Y-1)$ belong to new-elements(1) U ... U new-elements(k-1).

Theorem 18: For an L_0 program, if Conditions 1 to 3 are true at some k, then the program is in steady-state at k.

Proof: (By induction on k). From Lemma 16, all members of new-elements(k) are of the form $X(k_X)$ and, in addition, $X(k_X+1)$ must be in new-

elements(k+1). To prove that the program reaches steady-state at k, we must show that new-elements(k+1) cannot have a scalar input or the output of a t-box or any element of the form Y(j+1) (where Y can be a stream input, or the output of a T-box) if Y(j) is not in new-elements(k). Let us refer to such an element in new-elements(k+1) as an *off-beat* element.

Suppose α is an off-beat element in new-elements(k+1). From Proposition 15, there must be some Om(k+1) in new-elements(k+1) such that all intermediate nodes on the path from Om(k+1) to α in the dependence graph of Om(k+1) are in new-elements(k+1). Moreover, Om(k+1) cannot be an off-beat element - if Om(k) is in minimal-inputs(k), then Condition 3 (1) ensures that Om(k+1) must be either in new-elements(1) U ... U new-elements(k-1) or in new-elements(k) - either way, if Om(k) is in minimal-inputs(k), then Om(k+1) cannot be in new-elements(k+1). Therefore, on the path from Om(k+1) to a, there must be some element $Y(k_Y+1)$ (which can be Om(k+1) itself) which is not an off-beat element and which depends directly on an off-beat element β (which could be α itself). Condition 3 (2) now assures us that β must have been in new-elements(1) U ... U new-elements(k), and hence, it cannot be in new-elements(k+1). Consequently, no off-beat element can be a member of new-elements(k+1).

To complete the proof, we must show that Conditions 1 to 3 hold at (k+1). Given the result that no off-beat element can be a member of new-elements(k+1), the proof is trivial and is left to the interested reader.

□

We now describe an algorithm that will use Conditions 1 to 3 in order to compute the value of k at which an L_0 program reaches steady-state. It also generates the transformed program.

3.5 Algorithm-SST

Algorithm-SST : An algorithm to transform L_0 Programs into the Schema of Figure 5.

1. Set k to 0.

2. Set k to k+1 and compute new-elements(k) and minimal-inputs(k). Generate code for new-elements(k).

3. If Condition 1 holds for new-elements(k) then continue else go to 2.

4. If Condition 2 holds for new-elements(k) then continue else go to 2.

5. If Condition 3 holds for minimal-inputs(k) then continue else go to 2.

6. Set k_0 to k. The code generated for new-elements(1),...., new-elements(k_0-1) and generate code for the steady-state part in the following way. Replace each t-box in the code for new-elements(k_0) with the corresponding T-box and label the output stream of T_X-box as X_s. We will refer to this code as the *acyclic steady-state program.*

7. For each element a in minimal-inputs(k_0) which is of the form Oi(k_0), there must be some $k_{Oi} > k_0$, such that Oi(k_{Oi}) is in new-elements(k_0). Connect (k_{Oi}-k_0) *cons* boxes in a cascade and connect Oi(k_0), ..., Oi(k_{Oi}-1) from the prelude as shown in Figure 13.

8. For each pair of elements (X(k_X),a) such that X(k_X) is in new-elements(k_0), a in minimal-inputs(k_0), and X(k_X) directly depends on a, do the following:

   Let a be the p^{th} argument of the function that computes X(k_X). In the dependency matrix, let the starred entry for the p^{th} argument of T_X be $Y(j)^*$, and let it occur in X(s). Let Y(k_Y) be the element of Y in new-elements(k_0). There are 4 different cases for code generation depending upon whether Y is an input stream or not, and whether s is larger or smaller than k_X.

   a. Y is an input stream and $s \leq k_X$: Feed Y through k_X-s+j-1 *rest* boxes and connect the output to T_X-box as shown in Figure 14(i).

   b. Y is not an input stream and $s \leq k_X$: Connect s-k_X+k_Y-j cons boxes in a cascade as shown in Figure 14(ii).

   c. Y is an input stream and $s > k_X$: Feed Y first through j-1 *rest* boxes and then through s-k_X *cons* boxes as shown in Figure 14(iii). The scalar inputs for *cons* boxes are specified by entries in X(k_X) to X(s-1) locations of the dependency matrix, and must be available from the prelude.

   d. Y is not an input stream and $s > k_X$: Let Y(k_Y) be the element of Y in new-elements(k_0) ($k_Y > j$). Connect s-k_X+k_Y-j *cons* boxes in a cascade as shown in Figure 14(iv).

   We will refer to the code generated in step 8 as the *steady-state program.*

9. For each output stream O, introduce k_0-1 *cons* cells connected in a cascade. Connect O(1) to the scalar input of the first *cons* cell, O(2) to the scalar input of

the next *cons* cell and so forth. If $O(k_0)$ is in new-elements(k_0) then connect the output of the T-box T_O in the steady-state program to the stream input of the last *cons* cell, otherwise connect the output of the *cons* cell whose scalar input is $O(k_0)$ to the stream input of the last *cons* cell. (The values of O(1) to $O(k_0\text{-}1)$ must have been generated in the prelude part).

□

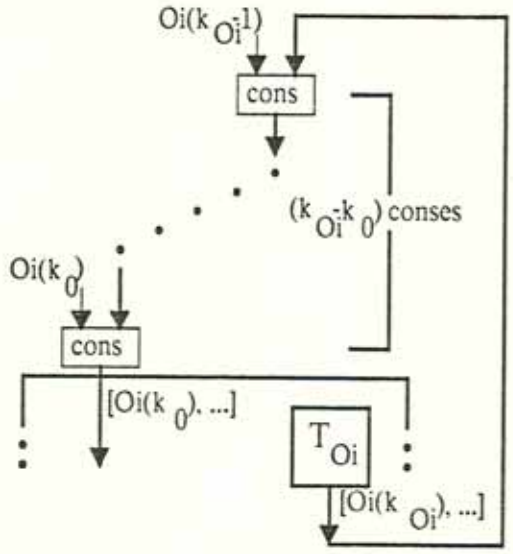
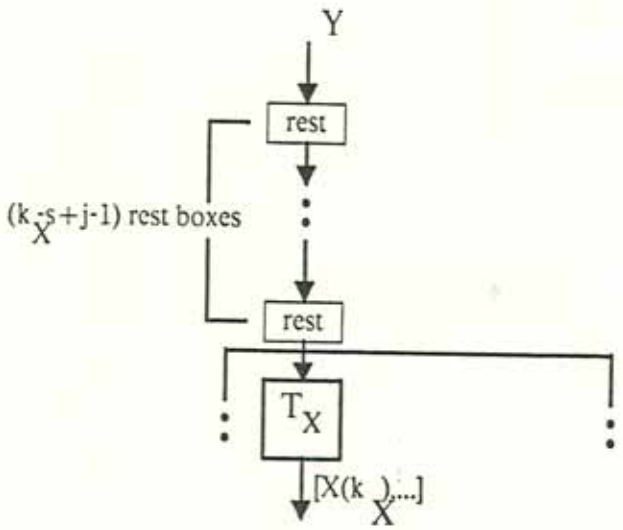


Figure 13: Generating Code for the Steady-State Part Related to an Output Stream

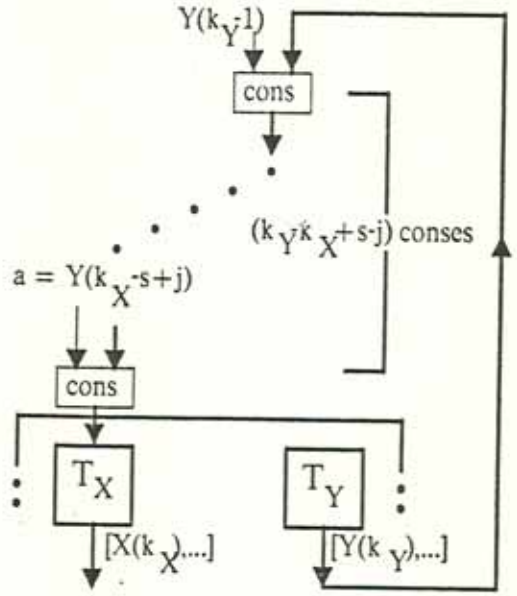
The behavior of Algorithm-SST when the algorithm is applied to the program of Figure 4 is summarized below.

k	Steps executed	
1	2,3	non-starred dependencies in new-elements(1)
2	2,3,4,5	nothing beyond C(1) or D(3) in new-elements(2)
3	2,3,4,5	nothing beyond C(2) in new-elements(3)
4	2,3,4,5,6,7,8,9	steady-state

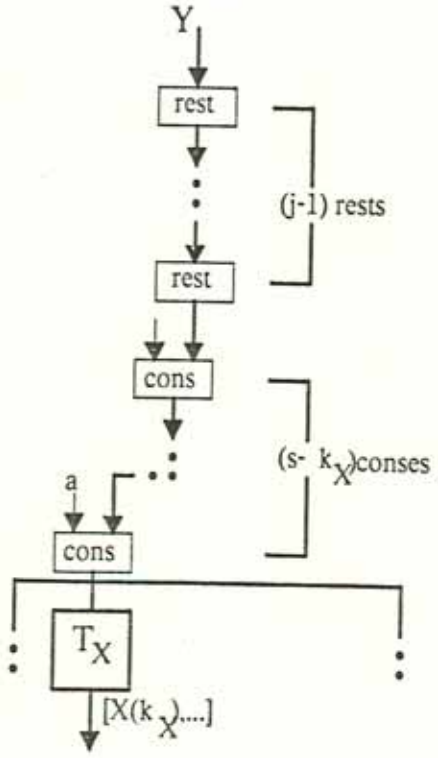
For the program of Figure 4, Step 8 of Algorithm-SST will examine the dependencies between C(3) and B(2), C(3) and D(4), and B(4) and D(3). The code generated by Step 8 will look like the code shown in Figure 11 with one minor difference - the *fork* shown at the output of the *cons* box with D(4) as an input will not exist. Instead, Step 8 will generate a third chain of *cons* boxes in which there is only one *cons* box; the scalar input to this *cons* box will come from D(4) while the stream input will be connected to T_D. The output of this



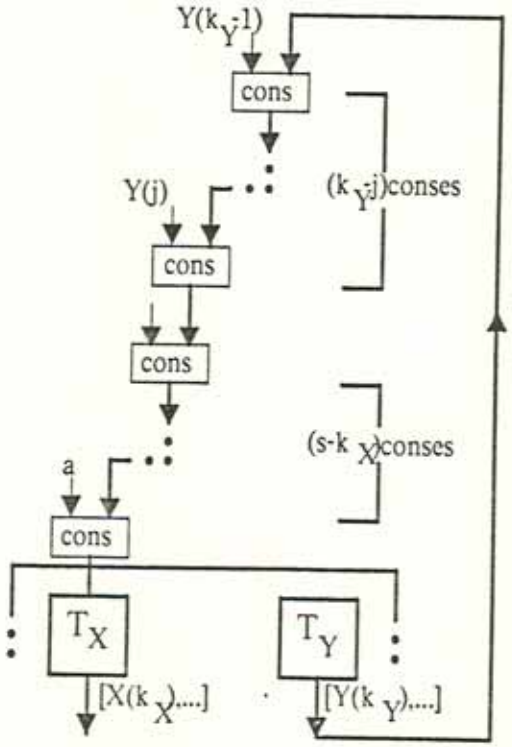
(i)Y is an input stream, $s \leq k_X$



(ii)Y is output of a T-box,$s < k_X$



(iii)Y is a stream input,$s > k_X$



(iv)Y is output of a T-box,$s > k_X$

Figure 14: Generating Code for the Steady-State Part

cons box will be connected to the inputs of T_C and T_D. It is not hard to modify Step 8 of Algorithm-SST so that it generates exactly the code shown in Figure 11; however, it is a lot easier to understand Algorithm-SST as it stands, and hence, the modified step is omitted.

The graph for the code generated for this example is shown in Figure 15.

Suppose in the program of Figure 4 we also declare B to be an output stream. The reader can verify that no change in new-elements will occur. In fact the only change will be in the value of minimal-inputs(2) where element B(2) will be included to be passed on as output at stage k=2 of the prelude. Consequently the behavior of Algorithm-SST does not change except for code generation at k=2. However, if O and C are designated as output streams then new-elements also change as shown below :

k	new-elements(O,k)	new-elements(C,k)
2	{O(2)}	{ }
3	{O(3),D(4),B(3)}	{C(3),D(4)}
4	{O(4),C(3),D(5),B(4)}	{C(4),D(5),B(4)}

k	new-elements(k)	minimal-inputs(k)
2	{O(2)}	{C(1),C(2),D(3)}
3	{O(3),D(4),B(3),C(3)}	{B(2),C(2),D(2),D(3)}
4	{O(4),D(5),B(4),C(4)}	{B(3),C(3),D(3),D(4)}

Algorithm-SST will still find the value of k_0 to be 4 but the code for steady-state part will change as shown in Figure 16.

We now prove the termination of Algorithm-SST. This proof is a little involved, and the reader can skip the rest of this section without loss of continuity. The difficulty in proving termination arises from the fact that Conditions 1, 2 and 3 are related in the sense that if Conditions 1 and 2 hold at some k_0, then Condition 3 must also hold at k_0 in order for Conditions 1 and 2 to hold for all $k \geq k_0$. An example which shows this connection is given in the appendix. Hence, we follow an entirely different proof strategy in which we first define two steady-state patterns - one for new-elements and the other for minimal-inputs. We then show that for sufficiently large k, new-elements(k) and minimal-inputs(k) of every L_0 program conform to these patterns. This will enable us to determine an upper bound for

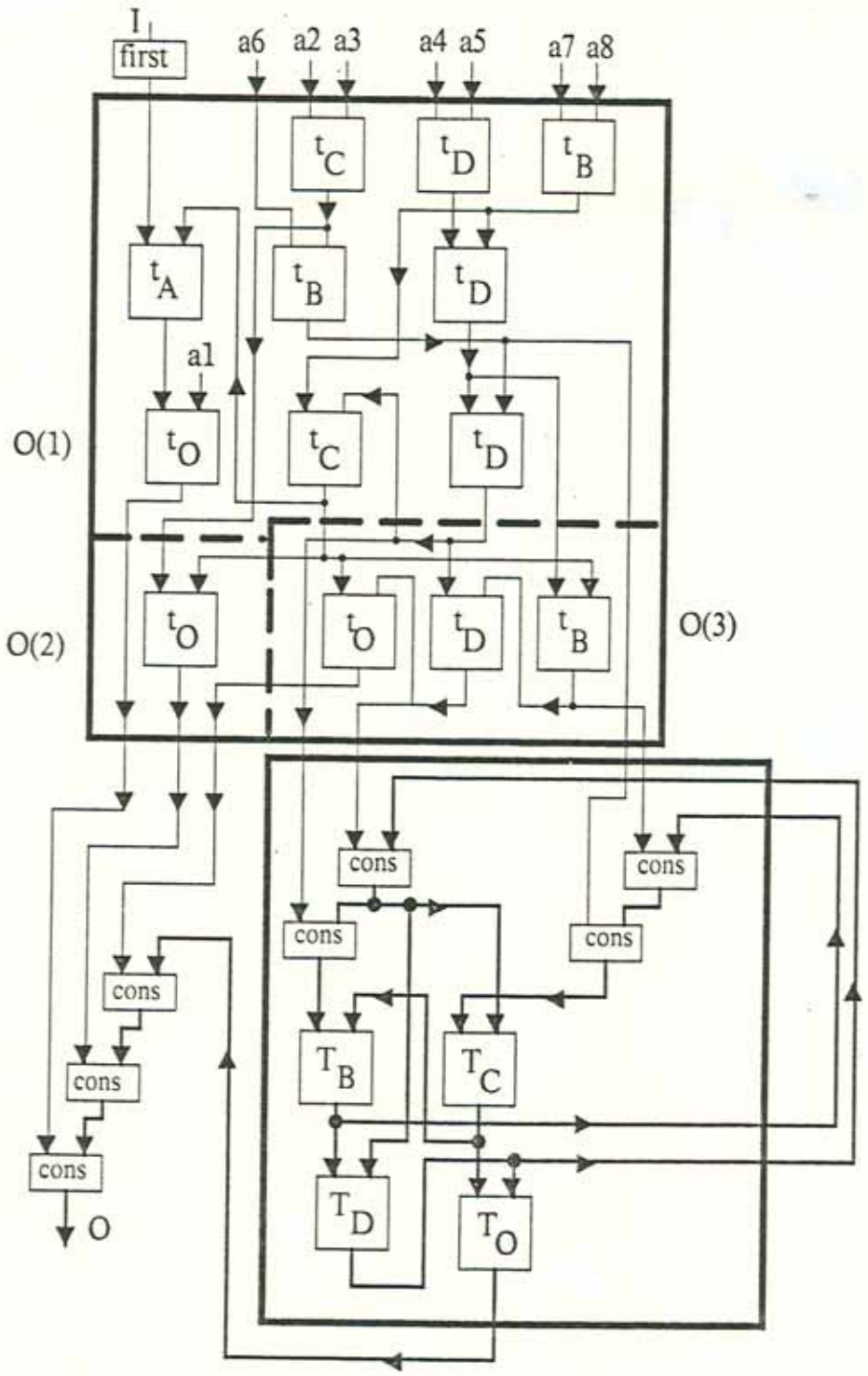


Figure 15: Transformation of program in Figure 4 by Algorithm-SST

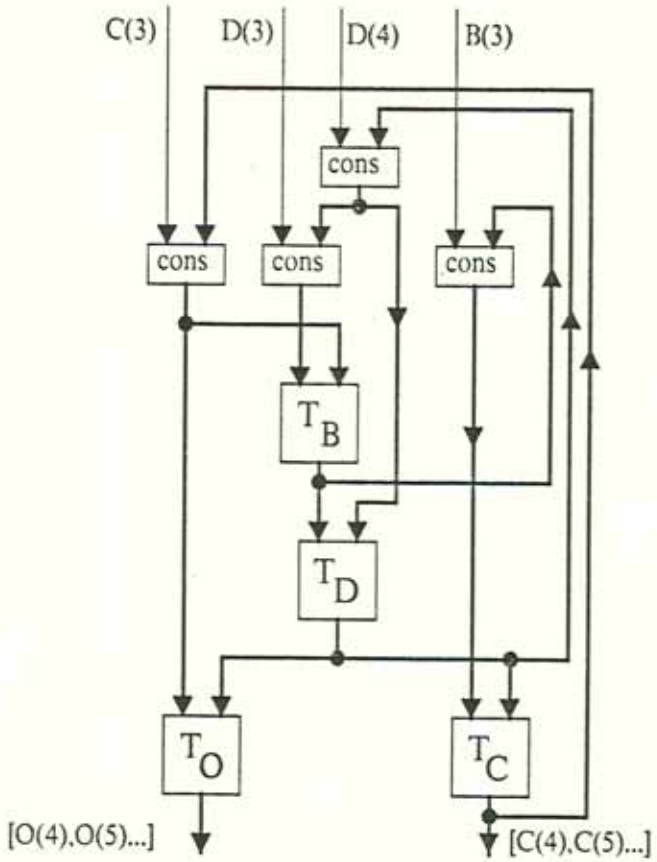


Figure 16: Steady-state Part of Program in Figure 4 if O and C are Outputs

the value of k_0. It is possible that the termination of Algorithm-SST can be proved in a more direct manner - however, we feel that the proof strategy given here is interesting in its own right. In order to define the steady-state patterns described above, we construct a weighted, directed graph for a data dependency matrix as follows.

Definition 19: Construct the *weighted graph* of a dependency matrix as follows:

1. For each stream input and T-box in the program, create a node and label it with the name of the input stream or T-box.

2. If $B(j)^*$ is a starred entry in position A(i) then create an arc from A to B and give it a weight of (j-i). Notice that, since for a large enough k, A(k) must be a starred element, it will depend upon B(k+j-i).

3. From the directed graph delete all nodes inaccessible from $\{Oi\}$, as well as any edges to and from these nodes.

The weighted graph for the program in Figure 4 is shown in Figure 17.

Lemma 20: If there is a path of weight w from some node A to some node B in the weighted graph, then, for sufficiently large k, A(k) depends on B(k+w).

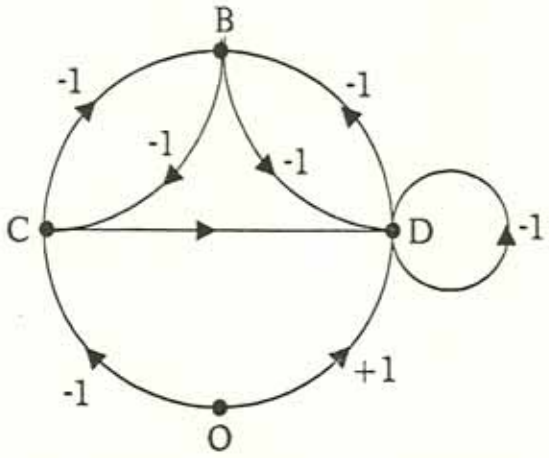


Figure 17: The Weighted Graph for the Program of Figure 4

Proof: This follows very simply from step (2) of the construction of the weighted graph. If the length of the path is 1, then for all k such that A(k) is a starred element, A(k) will depend on B(k+w). Suppose this is true for all paths of length n. A path of length n+1 between A and B will have some node C such that there is a path of length n from A to C and an arc from C to B. Let the weight of the path from A to C be w1 and the weight of the arc from C to B be w2. Since the path from A to C is of length n, A(k) will depend on C(k+w1) for all k greater than or equal to some kn. If C(kn+w1) is a starred element, then C(k+w1) (and hence, A(k)) will depend on B(k+w1+w2) for all $k \geq kn$. If C(kn+w1) is not a starred element, then suppose C(k) is a starred element for all $k \geq kc$ where kc must be greater than kn+w1. Then, A(k) will depend on B(k+w1+w2) for all $k \geq kc\text{-}w1$. Either way, since w1+w2 is the weight of the path from A to B, we have proved the required result.

□

For any output stream Oi, let the weight of the (trivial) path of length zero from Oi to itself be of weight zero. Let us now consider all possible paths from each element of {O*i*} to any node X, where X itself can be an element of {O*i*}. In general, a weighted graph will have cycles.(We do not consider the trivial path of length zero from Oi to itself to be a cycle.) However, the weight of a cycle (*i.e.*, the sum of the weights of all the arcs in the cycle) must be strictly less than 0. Otherwise, from the previous lemma, for some stream X and a sufficiently large value of j, X(j) will depend upon X(j+w) where $w \geq 0$, which will represent a deadlock. Since no cycle has a positive weight, and there are only finitely many output streams, there must be a path (from some output Oi to X) such that no other path from any output Oj to X has a weight larger than the weight of this path.

Definition 21: A *maximal path* to X is a path from some Oi to X such that no other path from any Oj to X has a larger weight.

In general, there can be more than one maximal path to X - for instance, if there is some path from Oi to X that is a maximal path, and there is a path of weight zero from Oj to Oi, then the path Oj...Oi...X is a maximal path to X that is, clearly, distinct from the path Oi...X. In our argument, we can use *any* maximal path to X.

The significance of maximal paths can be appreciated by looking at Figure 17. The maximal path from O to C has a weight of -1. In addition, there is a path O-C-B-C whose weight is -3. From Lemma 20, it follows that for a large enough k, O(k) will depend on C(k-1) and C(k-3). For a large enough k, C(k-3) is required for the computation of O(k-2), and hence it cannot be in new-elements(k). We will now formally prove this connection between maximal paths and new-elements.

The following definitions extract patterns from a weighted graph. We will subsequently relate these patterns to the steady-state of the program.

Definition 22: The set *pattern-elements*(k) is the set of elements of the form $X(k+w_X)$ where X is a node in the weighted graph, and w_X is the weight of the maximal path to X.

Definition 23: The set *pattern-inputs*(k) is the union of the set of all elements X(j+w) such that Y(j) is in pattern-elements(k) and there is an arc of weight w from Y to X , and X(j+w) is not in pattern-elements(k), with the set of all elements Oi(k) that are not in pattern-elements(k).

We encourage the reader to compare this definition to the definition of minimal-inputs(k).

Definition 24: The integer *gap* is the largest integer such that for some element $X(k+w_X)$ in pattern-elements(k), $X(k+w_X\text{-gap})$ is in pattern-inputs(k).

For the weighted graph of Figure 17, pattern-elements(k) is {O(k), B(k), C(k-1), D(k+1)} and pattern-inputs(k) is {B(k-2), D(k-1), D(k)}. These should be compared with new-elements(k_0) and minimal-inputs(k_0) computed by Algorithm-SST. Gap for this graph is 2 since D(k+1) is in pattern-elements(k) and D(k-1) is in pattern-inputs(k).

The reader may first want to read the next two lemmas and the theorem that follows them (without reading proofs) , in order to understand the line of argument for the proof of termination.

Lemma 25: There exists a k_1 such that, for all $k \geq k_1$,

1. pattern-elements(k) is a subset of new-elements(k), and

2. if all elements in pattern-inputs(k) are available, then new-elements(k) = pattern-elements(k).

Proof: We first compute the value of k_1 as follows:

Let us define a *starred element* to be an element all of whose immediate data dependencies are starred. Compute the set *transient elements* which is, the union of the transitive closures of the data-dependencies of all non-starred elements in the program. From Lemma 13, it follows that we can find a vector of integers j_A, j_B, ... so that $A(j_A)$, $B(j_B)$, ... are transient elements but $A(j_A+1)$, $B(j_B+1)$, ... are not. Let us refer to this vector as the *Transient Elements Vector* or TEV. k_1 is, then, the minimum value of k such that for every node X in the weighted graph, $k+w_X > TEV[X]$.

1. We first show that, for all $k \geq k_1$, every $X(k+w_X)$ in pattern-elements(k) must be in $\mathcal{T}(Oi(k))$ for some Oi. Let Oi be an output from which there is a maximal path to X, and consider a maximal path from Oi to X in the weighted graph. If A is an intermediate node on this path, then the path from Oi to A must be a maximal path to A. By the above definition of k_1, $A(k+w_A)$ be a starred element for all $k \geq k_1$. Since this holds for any node A in between Oi and X, Oi(k) must depend on $X(k+w_X)$.

Since $X(k+w_X)$ is in $\mathcal{T}(Oi(k))$, for all $k \geq k_1$, to show that it is in new-elements(k) we must prove that it could not have been computed during the computation of new-elements(1), ..., new-elements(k-1). If it had been computed, then let it be in $\mathcal{T}(On(j))$ $(1 \leq j \leq k-1)$. Consider the path in the dependence graph of On(j) from On(j) to $X(k+w_X)$. Either every node on this path is a starred element, or there is at least one non-starred element on this path. Since $k+w_X > TEV[X]$, the latter case is impossible. If the former is true, there must be a path in the weighted graph from On to X with a weight of $k+w_X-j$. Since $k-j \geq 1$, this weight is $\geq w_X+1$. This contradicts the fact that the maximal path to X is of weight w_X. Hence, $X(k+w_X)$ is in new-elements(k) for all $k \geq k_1$. A similar argument shows that $X(k+w_X+1)$, $X(k+w_X+2)$, ... cannot be in new-elements(1), ..., new-elements(k).

2. We now show that for $k \geq k_1$, if pattern-inputs(k) is available, then new-elements(k) = pattern-elements(k). If not, there must be some X(j) in new-elements(k) such that X(j) is not in pattern-element(k). Since we have assumed that pattern-inputs(k) are available this element can not be in pattern-inputs(k). Further, by the definition of new-elements, there must be some Oi(k) in new-elements(k) such that in the dependence graph of Oi(k), all intermediate elements (if any) between Oi(k) and X(j) must be in new-elements(k). Since Oi(k) is in pattern-elements(k), let Y(i) be the first element on the path that is neither in pattern-elements(k) nor in pattern-inputs(k) (of course, Y(i) may be X(j) itself), and let W(m) be the element on the path that depends on Y(i) (where W(m) may be Oi(k) itself). From the definition of Y(i), W(m) must be either in pattern-elements(k) or in pattern-inputs(k). We now have a contradiction - if W(m) is in pattern-elements(k), then Y(i) is in pattern-elements(k) or in pattern-inputs(k), while if W(m) is in pattern-inputs(k), then Y(i) cannot be in new-elements(k). Hence, if pattern-inputs(k) are available, then new-elements(k) = pattern-elements(k).

□

Lemma 26: For all $k \geq k_2$ where $k_2 = k_1 + \text{gap}$,

1. all elements in pattern-inputs(k) will be available,

2. minimal-inputs(k) = pattern-inputs(k),

Proof: 1. For some $k \geq k_2$, suppose $X(k+\alpha)$ is in pattern-inputs(k). By the definition of pattern-inputs, there must be some $X(k+\beta)$ in pattern-elements(k). Since $\beta-\alpha \leq$ gap, $k-(\beta-\alpha)$ is greater than k_1, and hence, $X(k+\alpha)$ must be in in the data-dependencies of some $Oi(k+\alpha-\beta)$. Therefore, for $k \geq k_2$, all elements in pattern-inputs(k) are available.

2. From Lemma 25 and part 1, it follows that for $k \geq k_2$, new-elements(k) = pattern-elements(k). The conclusion that pattern-inputs(k) = minimal-inputs(k) follows trivially from the definitions of pattern-inputs(k) and minimal-inputs(k).

□

Theorem 27: Algorithm-SST must terminate.

Proof: We have shown that for all $k \geq k_2$, pattern-elements(k) is equal to new-elements(k) and pattern-inputs(k) is equal to minimal-inputs(k). We will now show that the three conditions of Algorithm-SST are true at k_2.

Condition 1:We have shown that new-elements(k_2) = pattern-elements(k_2). For $k \geq k_1$, no transient element can be a member of pattern-elements(k). Hence, all dependencies between elements of new-elements(k_2) must be starred.

Condition 2: (Proof by contradiction) Let $X(k_X)$ be in new-elements(k_2) and assume X(i),for $i>k_X$ is in new-elements(1) U ... U new-elements(k_2). Hence, X(i) can not be in new-elements(k_2+i-k_X). However, it is in pattern-elements(k_2+i-k_X). This contradicts Lemma 26.

Condition 3: All elements in pattern-inputs(k_2) are of the form X(i) where X is a stream input or the output of a T-box. At k_2, all dependencies between elements of new-elements(k_2) and minimal-inputs(k_2) are starred. Hence, clause (2b) of Condition 3 does not apply at k_2. We will now show that clauses (1) and (2a) of Condition 3 are valid at k_2.

Suppose some element Oi(k) is in minimal-inputs(k). For each stream Oi, $Oi(k+w_{Oi})$ is in pattern-elements(k). Since (k_2 - k) must be less than gap, k must be $\geq k_1$, and hence, by Lemma 26, all elements of the form Oi(k+1), Oi(k+2) .... $Oi(k_2-1)$ must have been computed. Hence, clause (1) is satisfied.

The proof that clause (2a) of Condition 3 is satisfied at k_2 is exactly the same and is omitted. Hence, Algorithm-SST must terminate at some $k \leq k_2$.

□

The proof of correctness of the *code* generated by Algorithm-SST is omitted. One possible way of proving correctness is to show that the dependence set of any element Oi(k) of an output stream is exactly the same in both the untransformed and transformed programs. Since the code both in the prelude and in the steady-state program was generated directly from the dependency matrix of the untransformed L_0 program, the proof is straight-forward.

3.6 Discussion of Algorithm-SST

Condition 2 ensures that no stream element computed by the prelude is ever recomputed by the steady-state program. The reader can verify that if recomputation of stream elements is not forbidden, then the program of Figure 4 will reach steady-state at k =2. The transformed program would compute B(2), C(1), C(2) and D(3) twice - once in the prelude, and once in the steady-state program. On the other hand, the advantage of doing this is that the size of the prelude, and hence, of the transformed program, is reduced. Our own position on this trade-off is that the overhead of recomputation outweighs the benefits of a smaller program.

The reader can verify that if Clause (2b) of Condition 3 was omitted, then an L_0 program, in general, will reach steady-state at some $k \geq k_0$. An easy way of seeing this is to draw the dependency matrix of the transformed program, and apply Algorithm-SST to it. The reader can verify that if Clause (2b) was not there, the transformed program would get further transformed by Algorithm-SST. Clearly, this would not be a desirable situation.

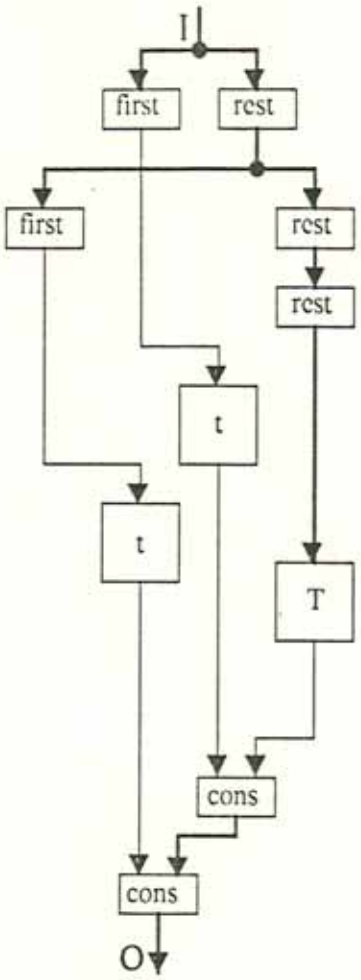
We would like to point out that the code for the steady-state program can be derived directly from the weighted graph. Once maximal paths have been determined in the weighted graph, we can easily determine pattern-elements and pattern-inputs. Unfortunately, it is not easy to determine the smallest value of k at which a program reaches steady-state. (The value of k_2 that was computed in the proof of termination of the L_0 program is an upper bound for the value of k at which the program reaches steady-state). Algorithm-SST requires more computation than this approach since we have to compute new-elements(1), ..., new-elements(k_0). On the other hand, the value of k_0 that it determines will, in general, be smaller than k_2. Moreover, we will use new-elements in the next section to propagate demands in the transformed L_0 program.

4 Making Transformed L_0 Programs Strongly-Safe

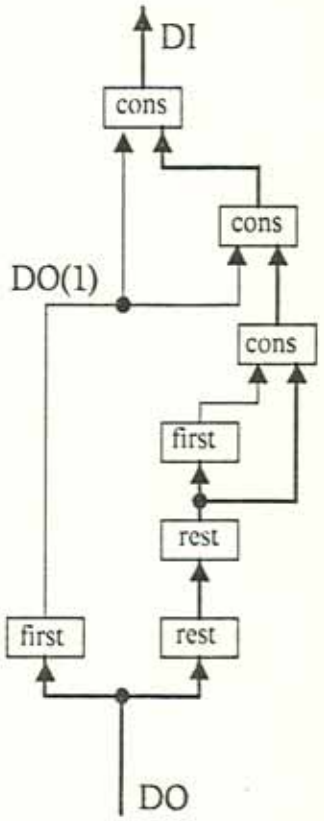
In this section, we make transformed L_0 programs that correspond to the schema of Figure 5 into strongly-safe programs that correspond to the schema of Figure 6. This is done in two steps - in the first step, we use a global algorithm for propagating demands for the outputs of the L_0 program to demands for the inputs of the program. In the second step, we introduce gates into the transformed program in order to make it safe. The resulting program is safe and input-output equivalent to its lazy program, and hence, it is strongly-safe (see Theorem 8).

4.1 Global Propagation of Demand Streams in L_0 Programs

Given the analysis of the last section, demand propagation in transformed L_0 programs is quite simple. Before giving the algorithm for demand propagation, we illustrate our technique by means of a simple example.



Transformed Program



Demand Propagation

I1 = *rest*(I);
O = T(*cons*(*first*(I1),*cons*(*first*(I),*rest*(*rest*(I1))))));

Figure 18: Demand Propagation in an L_0 Program

Consider the program shown in Figure 18. A demand for the first element of O must generate a demand for the second element of I. Our semantics for streams requires that the generation of I(2) be preceded by the generation of I(1). Consequently, the demand for O(1) must generate a demand for both I(1) and I(2). Consider now the demand for O(2). This demand can come only after the demand for O(1) - since the demand for O(1) resulted in demands for both I(1) and I(2), no new demands for the input need be generated. The program of Figure 18 reaches steady-state at k =3. It should be easy to see that the demand for O(3) should result in demands for I(3) and I(4). After k = 3, the pattern of demands is fixed - each request for an element of the output stream results in a request for one element

of the input stream. The reader can verify that the graph shown in Figure 18 generates precisely the pattern of demands for the input stream as discussed above. We now give an algorithm for generating the code for propagation of demands in an L_0 program. We will assume that the transformation of the source program has been done by applying Algorithm-SST.

Let a1, a2, ... be scalar inputs to an L_0 program, and I1, I2, ... be stream inputs to the program. As before, O1, O2, ... will represent the output streams of the L_0 program. Let DO1, DO2, ... represent demand streams for O1, O2, ... respectively. The algorithm first generates code to convert demand stream for each output stream into separate demands for scalar and stream inputs. In the next step, demands thus generated for an input are combined into a single scalar demand or a single demand stream depending upon the type of input.

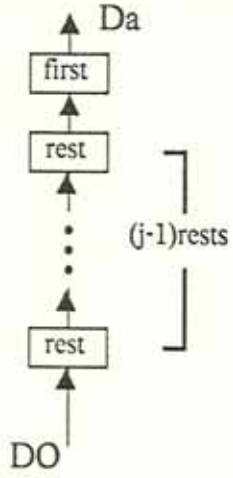
Algorithm-GDP: An algorithm for Global Demand Propagation in Transformed L_0 Programs.

1. For each output stream O in $\{Oi\}$ and each scalar input a, the code for generating the demand for a from the demand for O is produced as follows. If a does not occur in the sets new-elements(O,1), ..., new-elements(O,k_0) then there is no code to be generated for this pair of output stream and input. Otherwise, let j be the smallest integer such that a is an element of new-elements(O,j). Intuitively, this means that a is not required for the computation of O(1), ..., O(j-1), but will be required for the computation of O(j). The code for this case is shown in Figure 19(i).

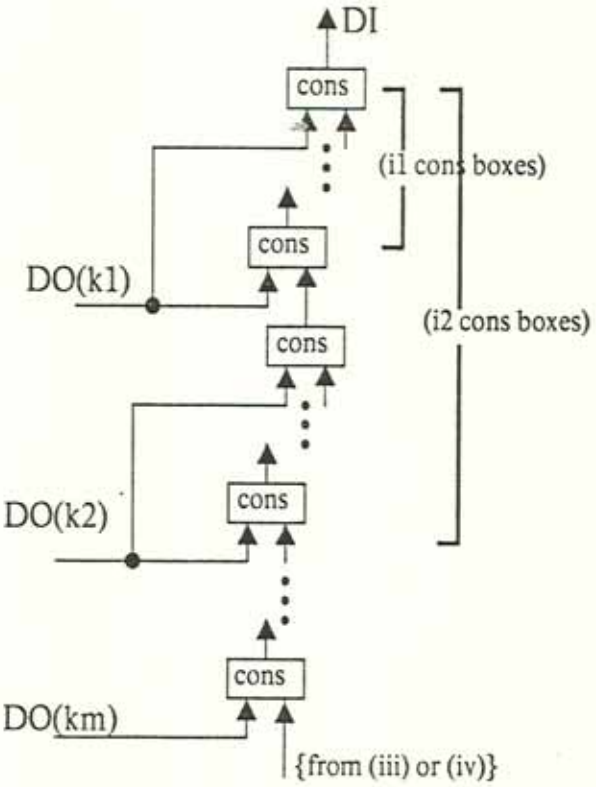
2. For each output stream O and each input stream I, do the following two steps:

   a. Examine new-elements(O,1), ..., new-elements(O,k_0-1) in increasing order of k, and form the sequence of pairs (I(i1),k1), (I(i2),k2), ... such that I(ij) occurs in new-elements(kj). Delete any pair (I(ij),kj) from this sequence if it is preceded by some pair (I(i),k) such that i$\geq$ij. In the remaining sequence $\{$(I(ij),kj)$\}$, the integers i1, i2, ... will form an increasing order. Let im be the largest integer in this sequence. The code to be generated is, then shown in Figure 19(ii).

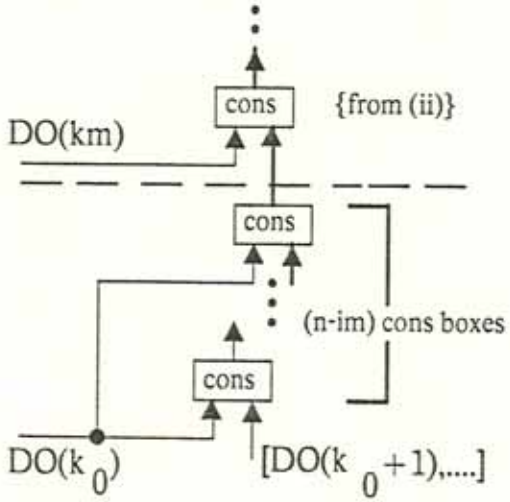
   b. If no element of I occurs in new-elements(O,k_0), there is no code to be generated. Otherwise, let n be the largest integer such that I(n) is in new-elements(O,k_0). If n > im then the code to be generated is shown in



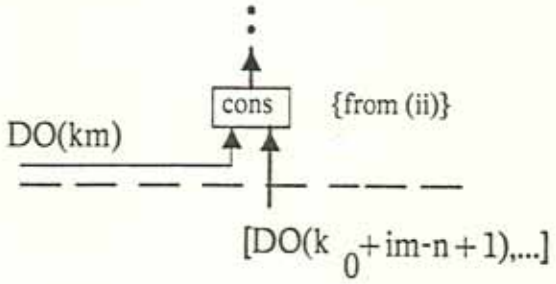
(i) Demand for an Elementary Input



(ii) Demand from Prelude for a Stream Input



(iii) Demand from Steady State (n > im)



(iv) Demand from Steady State (n < im)

Figure 19: Introducing Demand Streams into L_0 Programs

Figure 19(iii). Otherwise n $\leq$ im and the code should be generated according to Figure 19(iv).

3. For every scalar input, use a *d-union* to combine all the demands for its value. For every stream input, use a *D-union* to combine the demands propagated to it from the various outputs.

□

As before, Algorithm-GDP can be improved to yield better code - once again, we have not bothered to do this because we are interested in presenting our method in a simple way.

The final step in making a transformed L_0 program into a program that is input-output equivalent to its corresponding lazy program is to introduce gates at the outputs of the program. Consider the program shown in Figure 15. A demand for O(1) will cause the values of I(1) and a1, a2, ... , a8 to be fed into the program. Once these values are available, O(1) will be computed in the prelude, but so will the values of the O(2) and O(3). Moreover, this will also start the computation of O(4), O(5), ... in the steady state part of the program. In other words, a demand for O(1) will result in the production of all the elements of O. To ensure that the input-output behavior of the transformed program is the same as that of the corresponding lazy program, we must put a gate at every output of the transformed program by using the following algorithm -

Algorithm-IOE: Introducing Gates to Ensure Input-output Equivalence with Lazy programs.

For each output O*i*, introduce a gate on the line that produces the stream O*i* and feed the demand stream for O*i* into the control input of that gate.

□

Lemma 28: The transformed LD program that results from applying Algorithm-IOE has the same input-output behavior as the corresponding lazy program.

Proof: Omitted.

□

4.2 Introducing Gates to make L_0 Programs Safe

The final step in the transformation is to ensure that the transformed program will always perform at most a bounded amount of computation more than that performed by the corresponding lazy program. A program that results from applying Algorithm-IOE does not necessarily satisfy this requirement for the following reason. Since adjustments and prelude portions of the transformed program are acyclic interconnections of *first, rest, cons* and t-boxes, there cannot be an arbitrary amount of computation performed there if no input has an arbitrary number of tokens in it. Since inputs are fed in on demand, an input stream can have an arbitrarily large number of tokens in it if and only if the demand stream for some output has an arbitrarily large number of tokens in it. However, in that case, the corresponding lazy program will also perform an arbitrarily large amount of computation. On the other hand, the steady state program is, in general, a cyclic interconnection of operators. Therefore, operators in the steady state program can be enabled arbitrarily often even if there is no arbitrarily large demand for any output stream, as in the program shown in Figure 15. To ensure safety, we must, therefore, introduce gates into the steady state program. One way to do this is given in Algorithm-Safe.

Algorithm-Safe: Introducing Gates into the Steady-state portion of Transformed L_0 Programs.

1. Introduce a gate at the output of every *cons* chain that is generated by steps 8(b) and 8(d) of Algorithm-SST.

2. Use D-union operators to combine the demand streams [DO1(k_0), DO1(k_0+1), ...], [DO2(k_0), DO2(k_0+1), ...] etc. and feed the output of the D-union operator to the control input of each gate introduced in Step 1.

□

Theorem 29: The resulting program after Algorithm-Safe is safe.

Proof: To complete the proof of safety, we must show that no operator in the steady state program can be enabled arbitrarily often unless there is unbounded demand for some output stream. If there is no unbounded demand for any output stream, then no gate operator in the transformed program can be enabled an arbitrarily large number of times. Hence, no stream input or the output of any gate in the steady state program can have an unbounded number of tokens in it. Therefore, no operator in the acyclic steady state program can be enabled arbitrarily often. Therefore, the program that results from Algorithm-Safe is safe.

□

Algorithm-Safe introduces one gate for each cons chain (in the steady state program) whose stream input is connected to the output of some T-box in the acyclic steady state program. In many programs, the number of gates can be decreased still further by using the connectivity of the acyclic steady state program. For example, the transformed program in Figure 15 can be made safe by the introduction of only one gate, as shown in Figure 20. In Figure 20, box T_B in the acyclic steady state program cannot be enabled arbitrarily often (unless there is unbounded demand). This guarantees that there cannot be unbounded input to box T_C which makes it unnecessary to have a gate at the input of box T_C. Unfortunately, it can be shown that the problem of determining the smallest number of gates that must be introduced into an L_0 program in order to make it safe is equivalent to the vertex covering problem in graph theory - this problem is known to be NP-complete. Of course, in any practical problem, the number of inputs to the acyclic steady state program is likely to be quite small, and hence, enumeration of all possibilities for the introduction of gates may be quite acceptable.

Algorithm-Safe takes advantage of the fact that T-boxes are total functions in order to reduce the overhead of demand propagation. Notice that the control for gates permits one token to flow down each input line of the acyclic steady state program each time some element of an output stream is demanded. This is done even if the computation of that element of the output stream does not require a value on each input line. To look at it another way, a data-driven evaluation of the program resulting from Algorithm-Safe may perform a little more computation than a strict demand-driven evaluator would. However, this causes no problems with termination because, as we said before, T-boxes are total functions. The advantage of doing this is that the overhead of demand propagation has been lowered considerably.

Figure 20 shows a safe version of the program of Figure 4.

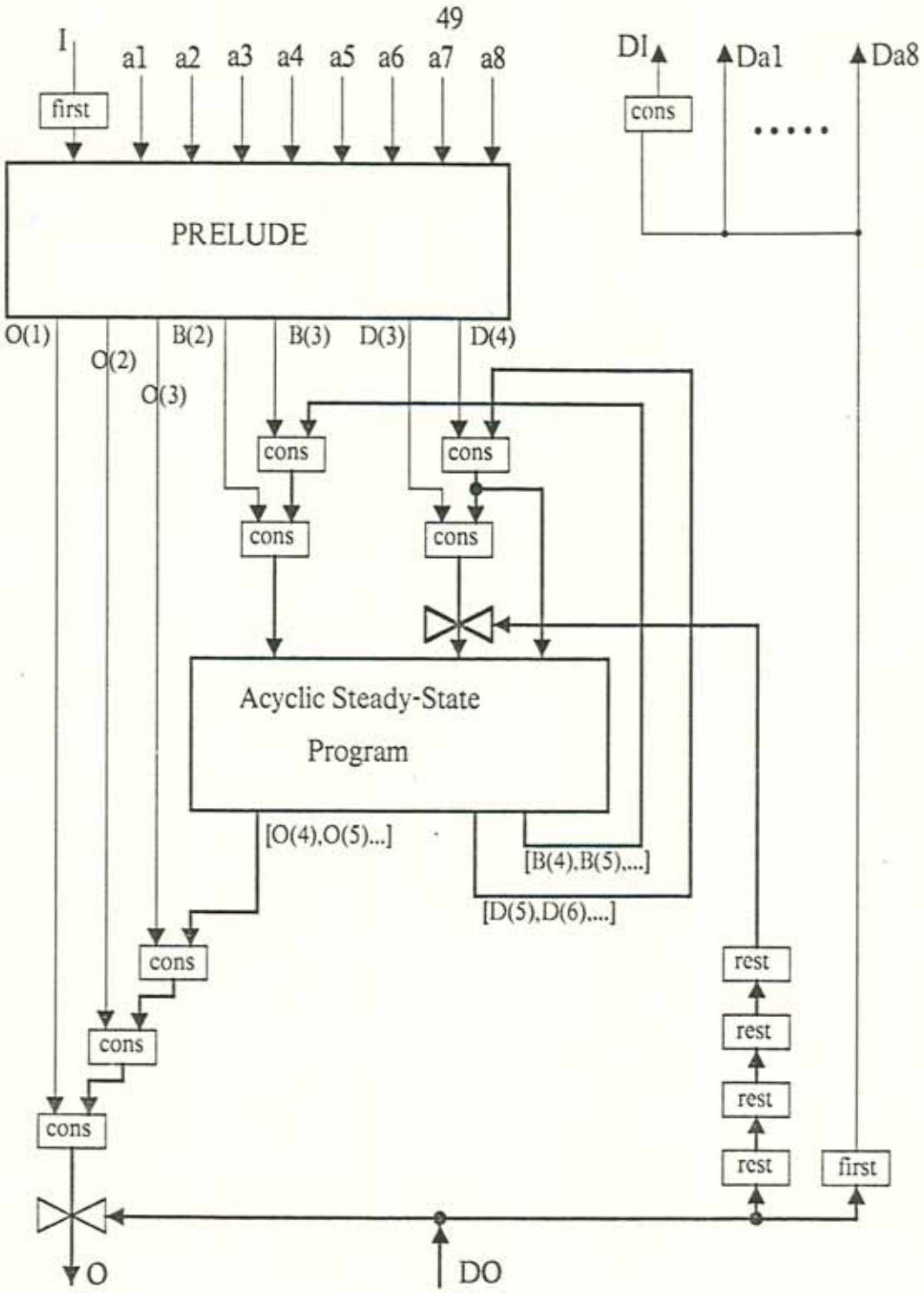


Figure 20: Strongly-Safe Version of Program in Figure 4

5 Summary and Further Work

In this paper and the companion paper, we defined a powerful stream processing language L and a super-set of L called LD which was the target language of transformations performed by the compiler. In paper 1, we gave a simple algorithm for transforming any L program into a program in LD (called the *lazy* LD program corresponding to the L program) which had the property that a data-driven evaluation of the lazy program performed precisely the same computation on data lines as a demand-driven evaluation of the original L program. This was proved by showing that lazy programs satisfied four properties named P1 to P4.

In this paper, we showed that a lazy program can be associated with any LD program and explored the concept of *safe* LD programs which were LD programs that performed a bounded amount of extra computation over that performed by their corresponding lazy programs. Since the set of safe programs is not closed under composition, we defined *strongly-safe* programs as being that subset of safe programs which is closed under composition. A class of strongly-safe programs is the set of all LD programs that are safe and input-output equivalent to their corresponding lazy programs.

We then defined a subset of L called L_0 and gave three algorithms for transforming any L_0 fragment of an L program into a strongly-safe program. This transformation was used to transform *any* L program into a strongly-safe program.

The work described here can be extended in many ways. The language L does not permit user-defined function calls. The transformational scheme in this paper can be applied without change to a language that has been augmented to permit user-defined function calls. An interesting observation is that many user-defined functions may behave like T-boxes in the sense that they have the *one-in-one-out* property. By treating such functions as T-boxes, it is possible to expand the scope of the transformational scheme to include such function calls in language L_0.

An interesting problem is to investigate how this work applies to languages with

generalized streams - *i.e.*, streams whose elements could be streams themselves. We feel that the view of a stream as a sequence of elements flowing down an arc in a dataflow graph must be abandoned in this case.

We have not supplied any performance measures that could determine if the advantage of reduced complexity of demand propagation is neutralized by the disadvantage of extra computation on data lines. Implementing the algorithms described in this paper could provide a clue.

We feel that efficient implementations of applicative languages are essential if such languages are to compete with imperative languages on sequential processors. The transformation described in this paper is only a beginning.

Acknowledgments: An earlier draft of this paper (without section 2 and under a different title) was submitted to TOPLAS. We are indebted to referee B for reading that version thoroughly and providing us with encouragement and detailed constructive criticism.

I. An Example

A	B(3),* B(5)*									
B	a	b	c	C(1)*						
C	d	e	A(1)	B(3)*						
O	C(1)	C(2)	C(3)	C(5)*						
	1	2	3	4	5					

Dependency Matrix for Example

k	new-elements(k)	minimal-inputs(k)	Test failed
1	{O(1),C(1),d}	{}	2
2	{O(2),C(2),e}	{}	2
3	{O(3),C(3),A(1),B(3),B(5),c}	{C(2)}	2
4	{O(4),C(5),B(4)}	{C(1)}	3
5	{O(5),C(6)}	{B(5)}	4
6	{O(6),C(7),B(6)}	{C(3)}	5
7	{O(7),C(8),B(7),C(4)}	{B(3)}	3
8	{O(8),C(9),B(8)}	{C(5)}	steady state

Figure I-1: An L_0 Program in which Conditions 1 and 2 are Related

References

1. Arvind, K. P. Gostelow, and W. Plouffe. Indeterminacy, Monitors and Dataflow. In *Operating Systems Review, Volume 11: Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, ACM-SIGOPS, 1977, pp. 159-169.

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

3. Pingali, K. and Arvind. Efficient Demand-driven Evaluation (I). Tech. Rep. ?, Laboratory for Computer Science, MIT, Cambridge, Mass., 1983.

4. Wadge, W. W. An Extensional Treatment of Dataflow Deadlock. In *Lecture Notes in Computer Science, Volume 70: Semantics of Concurrent Computation*, G. Kahn, Ed., Springer-Verlag, 1979, pp. 285-299.