

A BASIS FOR DATAFLOW COMPUTING

A.P.W. Böhm and J. van Leeuwen

RUU-CS-81-6

March 1981



Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

A BASIS FOR DATAFLOW COMPUTING

A.P.W. Böhm and J. van Leeuwen

Technical Report RUU-CS-81-6

March 1981

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

A BASIS FOR DATAFLOW COMPUTING*

A.P.W. Böhm and J. van Leeuwen

Department of Computer Science, University of Utrecht

P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Abstract. Current models of computation primarily are abstractions of traditional machine architectures. As recent advances in technology are leading to new principles for building machines, new principles for performing computations are recognized with it. Expanding on recent studies of Dennis et.al., we present a purified model of computation by dataflow as it appears on a lowest level of specification. Several connections to VSLI systems are indicated. We prove that for every partial recursive function f there is a dataflow net N that computes f and that has the additional property that it can be used for pipelining the computation of f -values on any sequence of inputs.

1. Introduction

Models of computation (see e.g. Minsky [10]) enable one to prove fundamental results about the power and limitations of real or proposed machine architectures. Much of the present theory of computation has resulted from the detailed analyses and abstractions of "von Neumann" architectures. As modern technology seems to be moving away from such traditional architectures, we need to revise our ideas about computation and the way it is performed accordingly. In this paper we shall explore the notion of computation by dataflow.

A dataflow computation is specified by a directed graph (a dataflow net) in which the nodes represent processing elements and the arcs represent data paths. A processing element can be a single instruction or a subprogram. There is no sequential flow of control in executing a dataflow computation. Instead, processing elements are activated by the arrival of their operands. Since many "instructions" may be activated by data at the same time, a high degree of parallelism can be achieved.

Using dataflow nets as "programs", a very different form of computer is

* A preliminary version of this paper was presented at the first meeting of the Working Community on Programming and Computer Architecture, Amsterdam, Nov. 6, 1980.

required to realize the intrinsic parallelism of execution in a suitable hardware organisation. Experimental dataflow computers are currently under construction at a number of institutions (including e.g. MIT, the University of Utah and Manchester University). Viewed from a different angle, there is nothing against letting processing elements be actual processors and arcs be wires. A dataflow net thus becomes the specification of an asynchronous design that may well be suited for implementation on a chip by means of current VLSI technology.

Dennis [4] has proposed a set of simple processing elements which he considers primitive to all dataflow computing. Fosseen [5] reportedly proved that these primitives indeed provide universal computing power. Recently Jaffe [7] extended the analysis of the framework proposed by Dennis, explored the connections with the theory of program schemata and proved the universality by simulating Turing machine computations in dataflow.

In Dennis' model a distinction is made between "control"-data and "data"-data, as two separate flows through the dataflow net. Corresponding to it, the primitive processing elements used have separate "gates" for control-data and data-data. In this paper we shall present a model in which this distinction is removed and only one, uniform type of data is used to drive all computation. The primitives we use are more elementary than those of Dennis and provide a very simple basis for dataflow computing. In sections 2 and 3 we shall introduce and explain the details of our model.

In Sections 3 and 4 we shall prove that our simplified model again has universal computing power in the sense of computability theory. Our proof is very different from Jaffe's [7] and shows direct constructions of dataflow nets for the primitive functions and standard operations from recursive function theory (see e.g. Rogers [11]). The constructions are of interest in view of the relation between dataflow computing and Backus' notion of functional programming ([2]), which derives much of its formalism from recursive function theory as well. Our main result will be that for each partial recursive function f there is a dataflow net to compute f that can be used for pipelining, i.e., for producing a continuous stream of outputs (f -values) corresponding to a continuous stream of inputs (argument values) without the need to ever reinitialize the net in between. Several applications of this result will be given in section 5.

2. Dataflow nets

In its most primitive form, a dataflow net is a directed graph in which the nodes represent processing elements and the edges represent datapaths. Some datapaths will not explicitly start at a node (the input-lines of the

net) and some will not explicitly end at a node (the output-lines of the net).

Data is presented in "tokens". Tokens are indivisible, but can be distinguished through an interpretation. In this paper we shall assume that all tokens are natural numbers. Tokens can be transmitted over existing datapaths. Processing elements digest tokens from their incoming edges and emit (send) new tokens over their outgoing edges. One cycle of a processing element normally consists of the consumption of one token from each incoming edge, followed by the production (and subsequent emission) of one token on each outgoing edge. The execution of a cycle is very similar to a "firing" in the terminology of Petri-nets. Processing elements are operators, i.e., fixed token-mappings of some variety. Except that cycles and token-transport take finite time, no further assumptions are made whatsoever about the speeds or relative speeds of the processing elements (cycle-time) or when processing elements choose to take in a next batch of tokens (initiate a new cycle). Dataflow computation is completely asynchronous. It implies that tokens may have to queue up along a datapath, if the node at the other end is not processing fast enough. In some models no queueing is actually permitted and processing elements will not "fire" unless the outgoing edges are free.

The many options in specifying a dataflow net have led to a number of different models, as shown in figure 1. Note that in all dataflow models,

INSERT FIGURE 1 ABOUT HERE

except in Kahn's [8] and Wadge's [13], the processing elements are assumed to be "token-level functional". It means that, given the same tokens on its incoming edges, an operator will always produce the same tokens on its outgoing edges, independent of the relative times of arrival of incoming tokens and of the state of the computation. Since dataflow computations are asynchronous, no functionality is guaranteed at the "global" (input/output) level unless proven.

Data is initially presented to a net by sending the appropriate tokens over the input-lines. From such a moment onwards, the computation is driven by the "flow of data" rather than by some kind of explicit "flow of control". Output (as tokens) will eventually be emitted over the output-lines. In the precise model used in this paper processing elements are allowed to fire

(i.e., execute a next cycle) only when all incoming edges have at least one token, with one well-defined exception: the JOIN-operator (see section 3). Tokens may queue, if needed. If they do, then a processing element will always pick the front element from each queue on its incoming edges, when it starts up a next cycle. The dataflow nets that will be designed in this paper actually will operate with queue-sizes restricted to 1.

Definition. A dataflow net is said to compute a (partial) function $f: \mathbb{N}^k \rightarrow \mathbb{N}$ when for all $x_1, \dots, x_k \in \mathbb{N}$ the net, upon receiving tokens representing x_1 to x_k over distinguished input-lines, will eventually output one token v if and only if $f(x_1, \dots, x_k)$ is defined and $f(x_1, \dots, x_k) = v$.

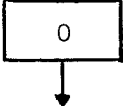
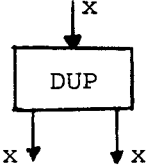

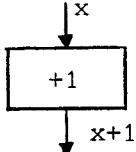
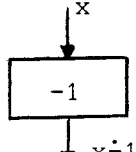
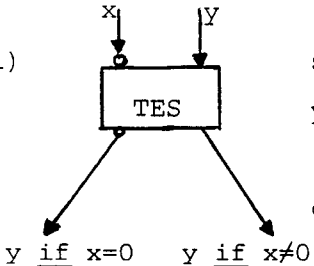
Given a dataflow net N computing some function f , it is to be expected that tokens will be left in the net even after the output $f(x_1, \dots, x_k)$ is produced. It means that N must be "cleaned" or re-initialized when it is to be used for another computation of an f -value, as otherwise the data left from the previous computation would foul up the "program" which, after all, is completely data-driven.

Definition. A dataflow net computing a (partial) function f is said to be weakly pipelined if the net does not have to be reinitialized (i.e., cleaned) whenever a next f -value is to be computed after a previous computation has ended. A dataflow net computing a (partial) function f is said to be pipelined if, upon receiving any continuous stream of k -tuples $\tilde{x}^1, \tilde{x}^2, \dots$ for which f is defined, the net will output a stream of values $f(\tilde{x}^1), f(\tilde{x}^2), \dots$ (in this order) without the need to ever reinitialize the net.

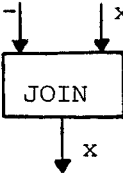
While the idea of pipelining computations certainly is not new (see e.g. [14]), no systematic study has apparently been made of the limits to pipelining in dataflow nets. Note that pipelined nets certainly are weakly pipelined. What functions can be computed (pipelined or not) will depend on the primitive operators chosen to build dataflow nets from. Traditionally, a set of primitives due to Dennis [4] is used. As we have simplified the model and eliminated control-flow signals as a distinguished type of data (as exploited by Dennis c.s.), a modified and more elementary set of primitives is required as a basis of dataflow computing. The next section will introduce and illustrate the primitives we propose.

3. Primitive processing elements and pipelining

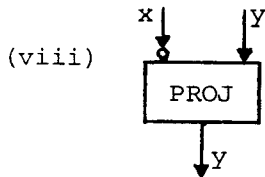
We shall use the following primitive processing elements ("boxes", operators) as ingredients for dataflow nets:

- (i)  specification: the 0-box emits a value (token) "0" once and is silent ever after.
- (ii)  specification: the DUP-box duplicates any incoming token (x) and emits a copy over either of its two outgoing edges.
- (iii)  specification: the SINK-box swallows and destroys any incoming token.
- (iv)  specification: the "+1"-box increments any incoming token (x) by 1 and emits the new value over its output line.
- (v)  specification: the "-1"-box decrements any incoming token (x) by 1, provided $x > 0$, and emits the resulting value over its output line. If $x = 0$, then the 0-value is passed on unchanged.
- (vi)  specification: upon receiving two inputs x and y on distinguished edges, the TES-box routes y "left" or "right" (i.e., on distinguished outgoing edges) depending on whether x equals 0 or not.

A very special box we include among the primitives is the following, which does not require that tokens are available on both its incoming edges before it actually fires. In fact quite the opposite is required: never shall there be a situation in which tokens are presented at both incoming edges simultaneously. (Note that this is very strict semantic constraint on the use of this box.)

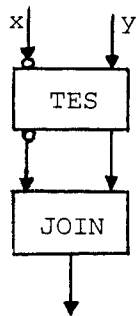
- (vii)  specification: the JOIN-box lets any incoming token pass through unchanged, provided it never finds tokens present on both incoming edges at the same time.

The constraint on the use of the JOIN-box is needed because, when two tokens would present themselves simultaneously, some decision would have to be taken as to what token should pass through first. Since no assumptions about cycle- and transport-times are made, the result of the JOIN-box would be as unpredictable (read: "non-functional") as the arrival times of the tokens, if the restriction were not made. For ease of use we shall introduce one more box, although it is not strictly independent of the primitives (i) to (vii):

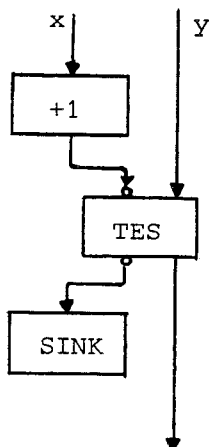


specification: upon receiving tokens x and y on distinguished input-lines, the PROJ-box will swallow x and pass on y over its outgoing edge unconditionally.

It would seem that the PROJ-box is merely a short-hand for the following "net":



but here we encounter a first fallacy in designing dataflow nets. When the JOIN-box is slow and the TES-box latches in pairs, say, $(0, y_1)$ and $(1, y_2)$ right after one-another, then the JOIN-box receives tokens y_1 and y_2 on its incoming edges simultaneously: a clear violation of the constraint on the use of JOIN-boxes! It is easily verified that the following graph is a correct net, realizing the task of the PROJ-box:



Since the TES-box is guaranteed not to receive a value 0 on its "x-port", the value y is passed on in the same manner as in the PROJ-box. It can be shown that the primitives (i) to (vii) are independent, considered as isolated graphs. In an operating environment one would not need (i) and could use any input and decrement it down to 0 to obtain a "one time 0" as desired, or allow initial values on the lines to start with (marking). In nets without inputs (generators, see section 5) the 0-box is needed, to start a computation.

The rules for building dataflow nets are straightforward. The notion of (asynchronous) computation by a dataflow net is identical to that for dataflow programs as in Dennis [4]. When nets are interpreted as VLSI-designs, the primitives and their connecting datapaths would have to be realized by means of transistors and wires, with drivers to transport signals (tokens) along the wires. In the theory of dataflow computing, however, the primitives are merely "transformers of data" that belong to an elementary repertoire, without reference to a specific machine or software implementation.

As an example, we will design a dataflow net for computing the function $f(x) \equiv 0$. Figure 2 shows three possibilities. The net in figure 2.a is

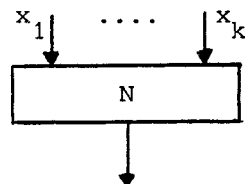
INSERT FIGURE 2 ABOUT HERE

formally correct, but is dead after firing once. The net in figure 2.b is a better try. It yields an output 0 whenever another input is presented, i.e., it is pipelined. But observe that there is no control that will prevent the continuous duplication of zeroes around the cycle in figure 2.b (even while no inputs are presented to the net) and, unless further precautions are taken, large queues of zeroes can form on lines A and B. Note that no queue of size > 1 would actually be required to let the net function correctly. Finally, figure 2.c shows a dataflow net to compute $f(x) \equiv 0$ in a pipelined fashion, that automatically avoids that there ever is a queue of size > 1 (except, perhaps, on the input line). An important problem in all later constructions is to actually prevent cycles from running out of control.

Figure 2.a has shown a very simple reason of why a net can fail to be pipelined. A second reason is that tokens, left behind from the computation on a previous set of arguments, provide an improper and incorrect "offset" for the computation on a next set of arguments. And a third reason is that

the next set of inputs may arrive "early" and mess up the entire ongoing computation. The first two reasons should be handled by a proper design, but for the third one can give a general construction to avoid it. Note that for a truly pipelined net one may want just the opposite, with no measures to actually avoid the opportunities for parallel computation. For later constructions, however, it will be needed that a net is not entered before the output from a previous "round" has been emitted.

Consider a dataflow net N computing a function f :



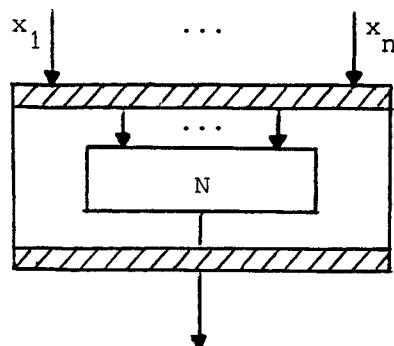
and assume that N is weakly pipelined. Our aim is to augment N to a fully pipelined dataflow net. To achieve it, we could surround N by a "sentinel" that will only let a next set of inputs through after a message is received that the output of the previous computation has been emitted. An attempt is made in figure 3. The sentinel construction given there is not

INSERT FIGURE 3 ABOUT HERE

necessarily right. When some input-token x_i was not consumed (hence, not needed) during a particular computation, it is not prevented from residing on the input-line until a next set of inputs arrives. If this happens, then it may get mixed into other argument sets than originally intended! This proves that (i) we must make sure that only nets are constructed which do use all tokens from a given set of inputs (or which ensure that unused tokens in one computation can only be routed to a safe place without harming any subsequent computations) and (ii) the sentinel construction abides by the same principle and forces that one token from every input-line is gated in with every next "round". (In the latter case, the net N will route the entire input-tuple consistently in the same manner.) The improved sentinel

INSERT FIGURE 4 ABOUT HERE

is shown in figure 4. Its correctness will be evident. Given a weakly pipelined dataflow net N , the augmentation with a sentinel construction as in figure 4 will be denoted as



The idea of strictly letting only entire input-tuples into a (sub-)net at any one time was implemented also by Rumbaugh [12] in his study of looping in dataflow programs.

Theorem 3.1. Let N be a weakly pipelined dataflow net for some function f and assume that N uses all input tokens, whenever it computes on a given set of arguments. The augmentation of N by the sentinel construction yields a fully pipelined dataflow net for f .

Proof

Consider figure 4. The construction guarantees that a next set of inputs is not gated in until the output from the previous computation with N finally appears on edge A . Since N is weakly pipelined, this forces a correct use of N , tuple after tuple. The same construction guarantees that, in order for the output of N to reach edge A , all input-tokens from the current set of inputs must have been gated in (along edge B , for every x_i). Since N uses all inputs, no input-token can stay behind and get mixed into new argument-tuples that it did not belong to. \square

The sentinel construction is intriguing because, when applied in the construction of larger nets, it essentially makes that tokens are pulled through by necessity, if a token is to appear on the output at all. Note that the net obtained by applying the sentinel construction again uses all input-tokens from a given set of arguments and thus it is ideally suited for use in further constructions where this property must be guaranteed.

4. Computing the partial recursive functions by dataflow

We assume that the reader is familiar with Kleene's characterization of the class of partial recursive functions (Kleene [9], Davis [3], Minsky [10], Rogers [11]). An inductive proof that every partial recursive function

can be computed by dataflow requires that we immediately prove the stronger result that every such function can be computed by a pipelined dataflow net. For when e.g. F is defined by primitive recursion from g and h :

$$F(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$

$$F(y+1, x_1, \dots, x_k) = h(y, x_1, \dots, x_k, F(y, x_1, \dots, x_k))$$

then a dataflow computation for F would naturally involve the pipelined use of a dataflow net for h . By theorem 3.1. it is sufficient that a weakly pipelined net for h (that uses all tokens from every input-tuple) is available. The problem in designing (weakly) pipelined nets is how one can ensure that nets are clean and ready for use after each iteration. It will appear that it does not really matter whether nets are left completely clean after each use on another tuple of arguments, as long as the values (tokens) that remain on the various edges do not interfere with any later uses of the net. Some constructions below are not spelled out in every detail.

Theorem 4.1. For every partial recursive function f there is a weakly pipelined dataflow net N that computes f . Moreover, N uses all input-tokens whenever it computes on a given set of arguments and automatically keeps the queue-sizes on its edges bounded by 1.

The proof proceeds by induction on Kleene's formation rules for the partial recursive functions. In the constructions below several typical problems in the design of dataflow nets will be highlighted.

(i) the constant-0 function $Z(x) = 0$.

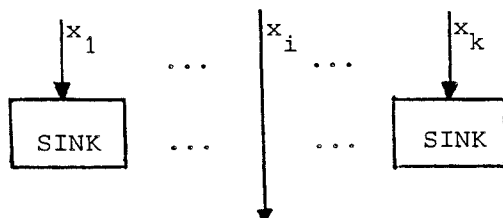
A net N to compute f according to the specifications of theorem 4.1. was given in figure 2.c.

(ii) the successor function $S(x) = x + 1$.

This function is trivially realized as required by just using the "+1"-box.

(iii) the projections $\pi_i(x_1, \dots, x_k) = x_i \quad (1 \leq i \leq k)$.

For any i ($1 \leq i \leq k$), π_i is realized as desired for theorem 4.1. by the following kind of dataflow net N :



N routes all "unused" arguments to sinks, which therefore never interfere with any other computation.

(iv) composition.

Let g be a partial recursive function of m variables and let h_1, \dots, h_m be partial recursive functions of k variables. Let F be defined by composition from g and h_1, \dots, h_m :

$$F(x_1, \dots, x_k) = g(h_1(x_1, \dots, x_k), \dots, h_m(x_1, \dots, x_k))$$

Suppose that g and h_1, \dots, h_m are computed by dataflow nets G and H_1, \dots, H_m respectively, which satisfy the requirements of theorem 4.1. It will be obvious that the net N shown in figure 5 satisfies the requirements

INSERT FIGURE 5 ABOUT HERE

as well and computes F .

(v) primitive recursion.

Let g be a partial recursive function of k variables and let h be a partial recursive function of $k+2$ variables. Let F be defined by primitive recursion from g and h :

$$\begin{aligned} F(0, x_1, \dots, x_k) &= g(x_1, \dots, x_k) \\ F(y+1, x_1, \dots, x_k) &= h(y, x_1, \dots, x_k, F(y, x_1, \dots, x_k)) \end{aligned}$$

Suppose that g and h are computed by dataflow nets G and H , respectively, which satisfy the requirements of theorem 4.1. It is much more involved this time to obtain a valid dataflow net N for F . We shall approach the construction in three stages.

Stage 1: route the input-tokens to G or to H , depending on the value of y .

The part of the construction that takes care of this is shown in figure 6 for the case $k = 2$. (For $k = 1$ or $k > 2$ the construction is adjusted in

INSERT FIGURE 6 ABOUT HERE

an obvious manner.) The R-graph will be specified later; it is the part of the net where the actual recursion for $y > 0$ will take place. For $y = 0$ all input-tokens will be gated to G, for $y > 0$ they will all be gated to the R-graph. It follows that for $y = 0$ the net N (as known up to this point) functions as desired, while for $y > 0$ there is no way the arguments can end up in this same part of the net. Note that the JOIN-box is used properly, since it can never occur that tokens come in from both directions (G and R) simultaneously, before the net would be used again. (This demonstrates that the sentinel construction of section 3 would actually be needed to preserve the wellformedness of this dataflow net in case it is pipelined, apart from the reasons we saw until now!) The difficulties all accumulate in the design of the R-graph.

Stage 2: implement the recursion in subnet R.

R will receive data only when $y > 0$. Its task is to compute and emit the value $F(y, x_1, \dots, x_k)$. The obvious idea is to compute it by generating

INSERT FIGURE 7 ABOUT HERE

the values $F(j, x_1, \dots, x_k)$ for j from 0 to y , through the pipelined use of H. The main part of the construction is shown in figure 7. Since H is weakly pipelined but used in a fully pipelined manner, it is surrounded by the sentinel-construction (cf. section 3). This will guarantee that it "pulls in" a full set of arguments for every next j . Some care must be exercised that the various "cycles" (the unspecified subnets in figure 7) do not run wild in generating next tuples of arguments for the recursion. In figure 7 this is arranged by letting H generate a signal whenever another $F(j+1, x_1, \dots, x_k)$ is produced. The signal is 1 or 0, depending on whether the final j -value ($j=y$) has been reached or not. The signal is gated to the various cycles. As long as the signal is 0, a next tuple of arguments is generated and gated towards H; this will involve incrementing j by 1 and reproducing every x_i . Whenever the signal becomes 1, the current j -value and the x_i 's are gated towards a sink. The signaling guarantees that the recursion is carried out a proper number of times. More importantly, it guarantees that no unnecessary tokens are generated (like j -values larger than y), that queue-sizes remain bounded by 1 and that all tokens are removed from the active parts

of the net (gated towards a sink) when the recursion is at an end. Provided the remaining parts of the net are correctly specified, R satisfies all requirements for being weakly pipelined! Note that R uses all its arguments by virtue of the fact that the G- and (pipelined) H-graphs do so.

Stage_3: wind up the remaining details.

It is intriguing to note in figure 7 that the JOIN-boxes are correctly used. In particular, there can be no delayed queueing on the incoming edges of the JOIN-box in the lower right corner, because the "signal" will be "pulled out" by all places that need it (which, in turn, are pulled by the H-graph which needs a complete set of arguments) every time through the recursion. All we need to do is supply the correct dataflow-logic for the unspecified subgraphs in figure 7. The constructions are all rather immediate

INSERT FIGURE 8 ABOUT HERE

and shown in figure 8.a through e. The easy verifications are left to the reader. Note that nowhere queue-sizes > 1 can occur (except perhaps at slow sinks).

(vi) minimization.

Let g be a function of $k+1$ variables. Let F be defined by minimization from g :

$$F(x_1, \dots, x_k) = \mu Y [g(Y, x_1, \dots, x_k) = 0]$$

(The μ -operator is to be interpreted as "the smallest such that".) Suppose that g is computed by a dataflow net G that satisfies the requirements of theorem 4.1. Again we shall construct a proper, weakly pipelined dataflow net for F in stages.

Stage_1: make a global design for N .

To compute F , we shall implement the straightforward idea of computing the values $g(j, x_1, \dots, x_k)$ for j from 0, until a value "0" is encountered. The construction of a dataflow net for it is shown in figure 9. Since G is obviously used in a pipelined fashion, it is surrounded by the sentinel

INSERT FIGURE 9 ABOUT HERE

construction (cf. section 3). This will guarantee that it swallows a full tuple of arguments every time around. As long as the g -value remains non-zero, a next j -value will be generated and gated to G , together with a next set of copies of x_1 to x_k . To keep the cycles in the net from running wild, we again use a signalling mechanism. After another g -value is generated it is tested. A signal will be set to 1 or 0, depending on whether the g -value is 0 or not. The signal is gated to all places that need it (using DUP-boxes, which are not shown in figure 9). When the signal is 0, it will trigger the generation of a next set of arguments of G . When the signal is 1, it will direct the current j -value and the cycling x_i -values to sinks and, thus, remove them from the net. At the same time, the current j -value is sent down the output line of the net as the result of the computation.

Stage 2: wind up the usual details.

It can be noticed that the ingredients to the current construction are very similar to that of (v). In particular, the unspecified subgraphs in figure 9 are identical to the corresponding graphs shown in figure 8. Observe again that queues remain bounded by 1 in size and that all tokens are directed to safe parts of the net (i.e., towards the sinks) when a current computation ends. N uses all arguments by virtue of the fact that (the pipelined version of) G does. Hence N is a weakly pipelined dataflow net for F as required for theorem 4.1.

This completes the proof of theorem 4.1. Together with 3.1. we can use immediately conclude our main result:

Theorem 4.2. ("the pipeline theorem") For every partial recursive function f there is a pipelined dataflow net N that computes f . Moreover, no queues in N need to have size greater than 1.

We note that, conversely, every dataflow net can easily be simulated by a nondeterministic Turing machine. The nondeterminism of the machine is needed to "guess" which boxes will fire at any particular moment (viz. which boxes will fire simultaneously). It follows that dataflow nets, as

defined in this paper, provide yet another basis for general computability theory. (Compare Jaffe [7], where this conclusion was proved for the dataflow primitives of Dennis [4].)

5. Some applications of the pipeline theorem.

From theorem 4.2. one can immediately derive a great many undecidability results for dataflow computing. We shall only mention one.

Definition. A dataflow net is said to be well-formed when (i) it is correct as a graphical structure and (ii) no occurring JOIN-boxes will ever receive tokens on both their incoming edges simultaneously in any computation.

Theorem 5.1. Well-formedness of dataflow nets is undecidable.

Proof

Suppose well-formedness were decidable. Consider dataflow nets of the sort as displayed in figure 10, where we allow f to be any partial recursive

INSERT FIGURE 10 ABOUT HERE

function. A net of this sort is well-formed if and only if f is everywhere undefined. Since the latter property is known to be undecidable, our assumption is contradicted. \square

An immediate conclusion is that wellformedness, like correctness, can only be ensured through a precise and disciplined construction-procedure for dataflow nets. There is a second conclusion to be drawn from 5.1. It can be argued that wellformedness and functionality of a dataflow net are, in a certain sense, equivalent concepts. Hence the functionality of a dataflow net is undecidable just like, as a matter of fact, the functionality of a nondeterministic Turing machine is undecidable.

Several further applications of the pipeline theorem relate to the generation of sets. Hitherto only a few examples were known (c.q. given) of dataflow nets which would emit "sequences of numbers of a specified kind (like prime numbers), in a specified order". Very generally we can now state the following theorem:

Theorem 5.2. For any recursively enumerable set S there is a dataflow net that generates and outputs the members of S in enumeration order. Moreover, the net does not need any queue-sizes to be larger than 1.

Proof

It is well-known that any non-empty rec. enumerable set S is the range of a total recursive function F (cf. Rogers [11], § 5.2.). Thus, to enumerate S by dataflow, all we need to do is feed the arguments $0, 1, 2, \dots$ into a pipelined dataflow net for F . The construction is shown in figure 11

 INSERT FIGURE 11 ABOUT HERE

and explicitly shows the sentinel construction, to emphasize that the net for F is pipelined. A next argument j is gated into the net whenever another element of S has been generated. The construction avoids that j -values queue up in quantities larger than 1 on the input-line of F .

Another well-known result (cf. Roger's [11], § 5.1.) states that non-empty rec. enumerable sets can be effectively enumerated in non-decreasing order. The result obviously carries over to the generation of sets by dataflow.

Finally, we formulate the existence of "universal" dataflow nets in a rather strong sense. Let $\{\varphi_z^{(k)}\}$ be a Gödel-numbering of the partial recursive functions of k variables.

Theorem 5.3. For every k there exists a dataflow net $N^{(k)}$ with $k+1$ input-lines, such that on input z, x_1, \dots, x_k the output is $\varphi_z^{(k)}(x_1, \dots, x_k)$. Moreover, $N^{(k)}$ can be pipelined and does not need internal queues of size greater than 1.

Proof

By Rogers [11], § 1.8., Thm IV there is a partial recursive function F of $k+1$ variables that is a universal partial function for the class of partial recursive functions of k variables, i.e., for all z and x_1 to x_k :

$$F(z, x_1, \dots, x_k) = \varphi_z^{(k)}(x_1, \dots, x_k)$$

(where both sides of the $=$ - sign are defined or undefined simultaneously).

$N^{(k)}$ is the dataflow net for F , as it is implied by the pipeline theorem. \square

$N^{(k)}$ may be termed a "general purpose dataflow computer". As a net, $N^{(k)}$ may not be practical for a number of reasons, including the lack of "optimum" parallelism resulting from the straightforward simulating of sequential computation and the prohibitive size of the numbers that can accumulate on the edges. Yet it shows the universality of dataflow computing at the lowest possible level.

6. References.

- [1] Arvind and K. Gostelow, Some relationship between asynchronous interpreters of a dataflow language, in: E.J. Neuhold (ed.), Formal descriptions of Programming Concepts, North-Holland Publ. Comp., Amsterdam, 1978, pp. 95-119.
- [2] Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, C. ACM 21 (1978) 613-641.
- [3] Davis, M., Computability and unsolvability, McGraw-Hill, New York, NY, 1958.
- [4] Dennis, J.B., First version of a data flow procedure language, in: Programming Symposium, Springer Lecture Notes in Computer Science 19, Springer Verlag, Berlin, 1974, pp. 362-276.
- [5] Fosseen, J.B., referred to in [4].
- [6] Gurd, J., I. Watson and J. Glauert, A multilayered data flow computer architecture, draft rep., University of Manchester, 1978.
- [7] Jaffe, J.M., The equivalence of r.e. program schemes and dataflow schemes, J. Comp. Syst. Sci. 21 (1980) 92-109.
- [8] Kahn, G., The semantics of a simple language for parallel programming, in: Information Processing 74, North Holland Publ. Comp., Amsterdam, 1974, pp. 471-475.
- [9] Kleene, S.C., General recursive functions of natural numbers, Math. Ann. 112 (1936) 727-742.
- [10] Minsky, M.L., Computation: finite and infinite machines, Prentice-Hall Inc., Englewood Cliffs, NJ, 1967.
- [11] Rogers Jr., H., Theory of recursive functions and effective computability, McGraw-Hill, New York, NY, 1967.

- [12] Rumbaugh, J., A data flow multiprocessor, IEEE Trans. Comp., C-26 (1977) 138-146.
- [13] Wadge, W., An extensional treatment of dataflow deadlock, in: G. Kahn (ed.), Semantics of concurrent computation, Springer Lecture Notes in Computer Science 70, Springer Verlag, Berlin, 1979, pp. 285-299.
- [14] Weng, K.-S., Stream-oriented computation in recursive data flow schemas, Techn. Memo 68, Lab. for Computer Science, MIT, Cambridge, Mass., 1975.

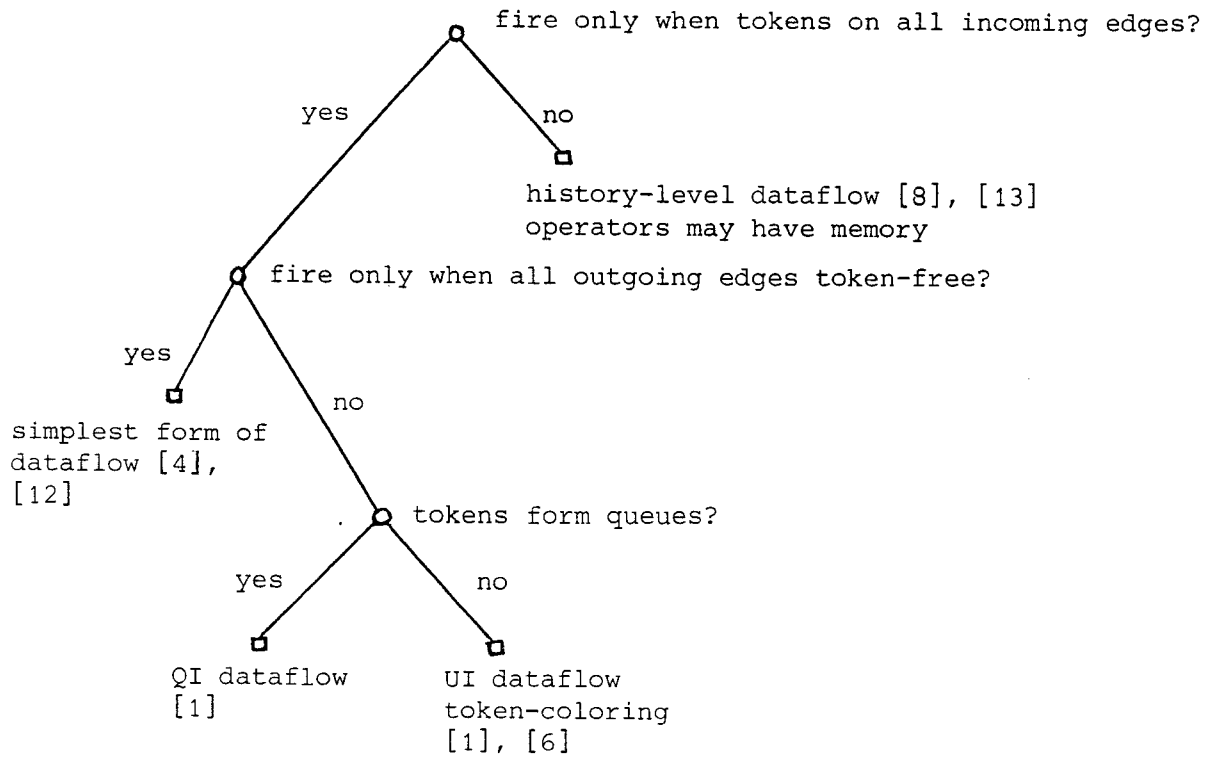
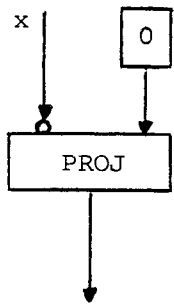
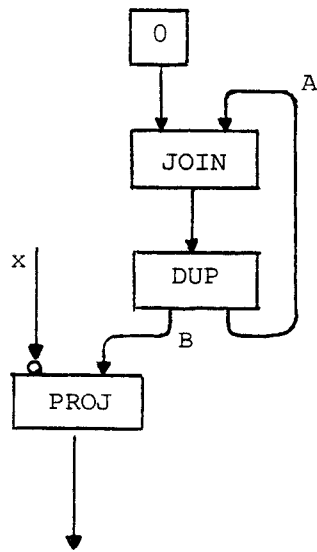


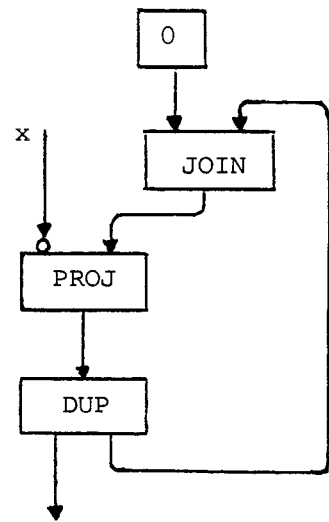
Figure 1. Choices in existing dataflow models



(a)



(b)



(c)

Figure 2. Computing $f(x) \equiv 0$.

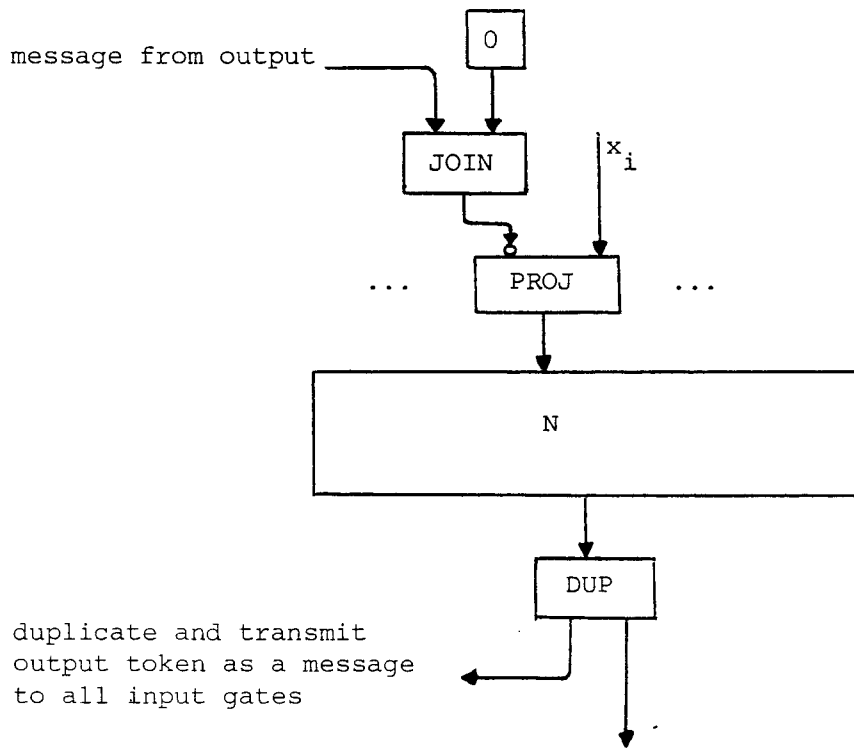


Figure 3. A failing sentinel construction.

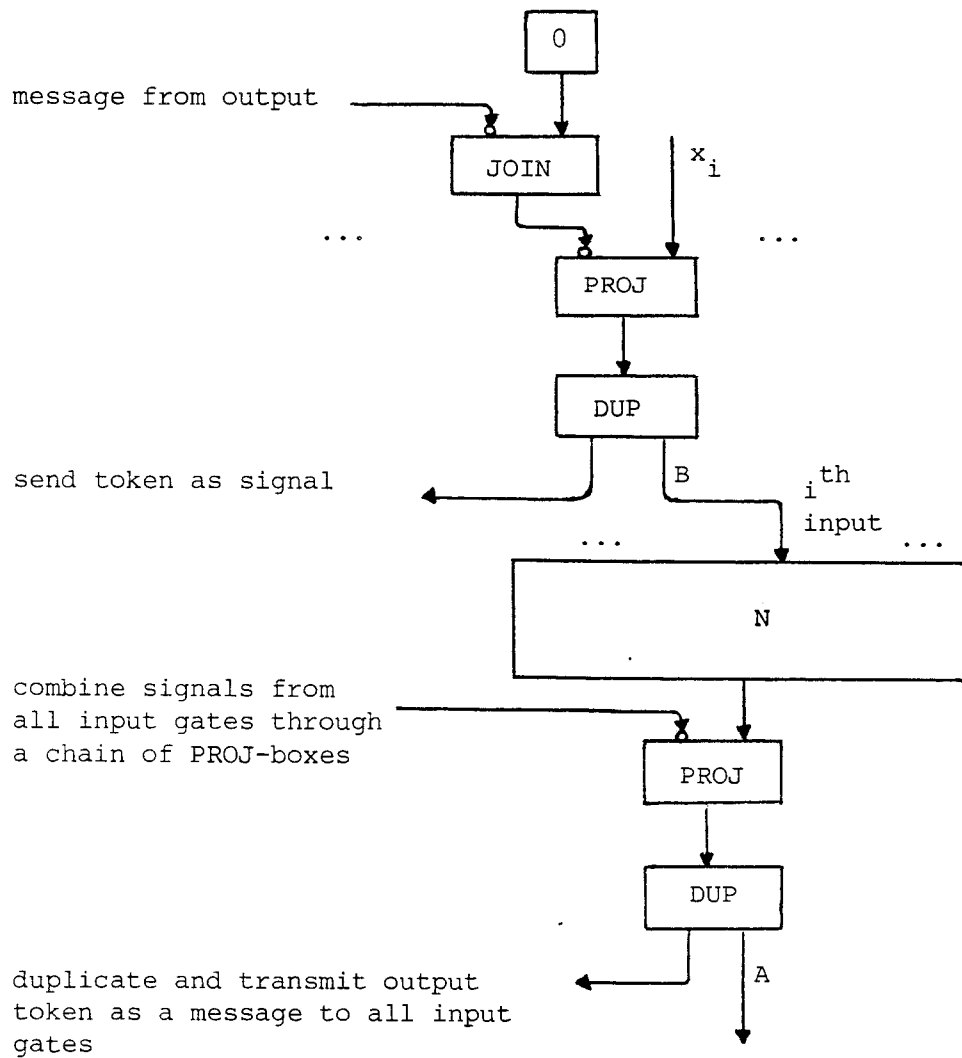


Figure 4. A valid sentinel construction

$x_1 \downarrow \quad \dots \quad \downarrow x_k$
duplicate tokens to all H_j -nets

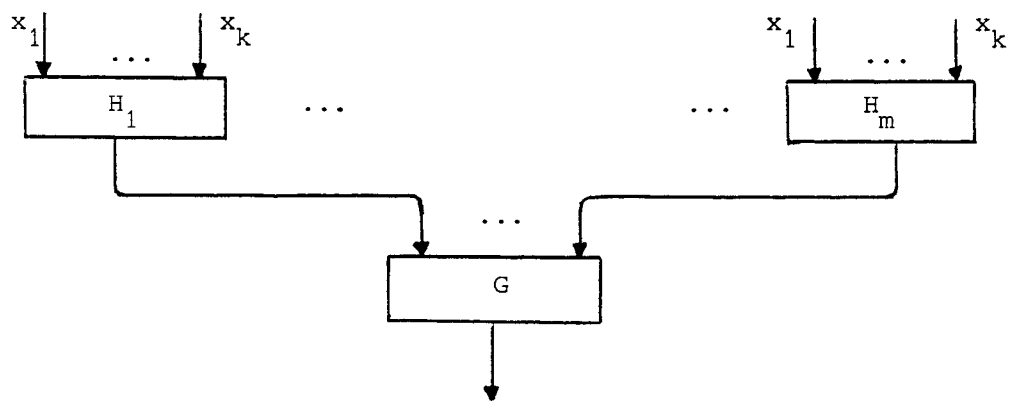


Figure 5. Composition

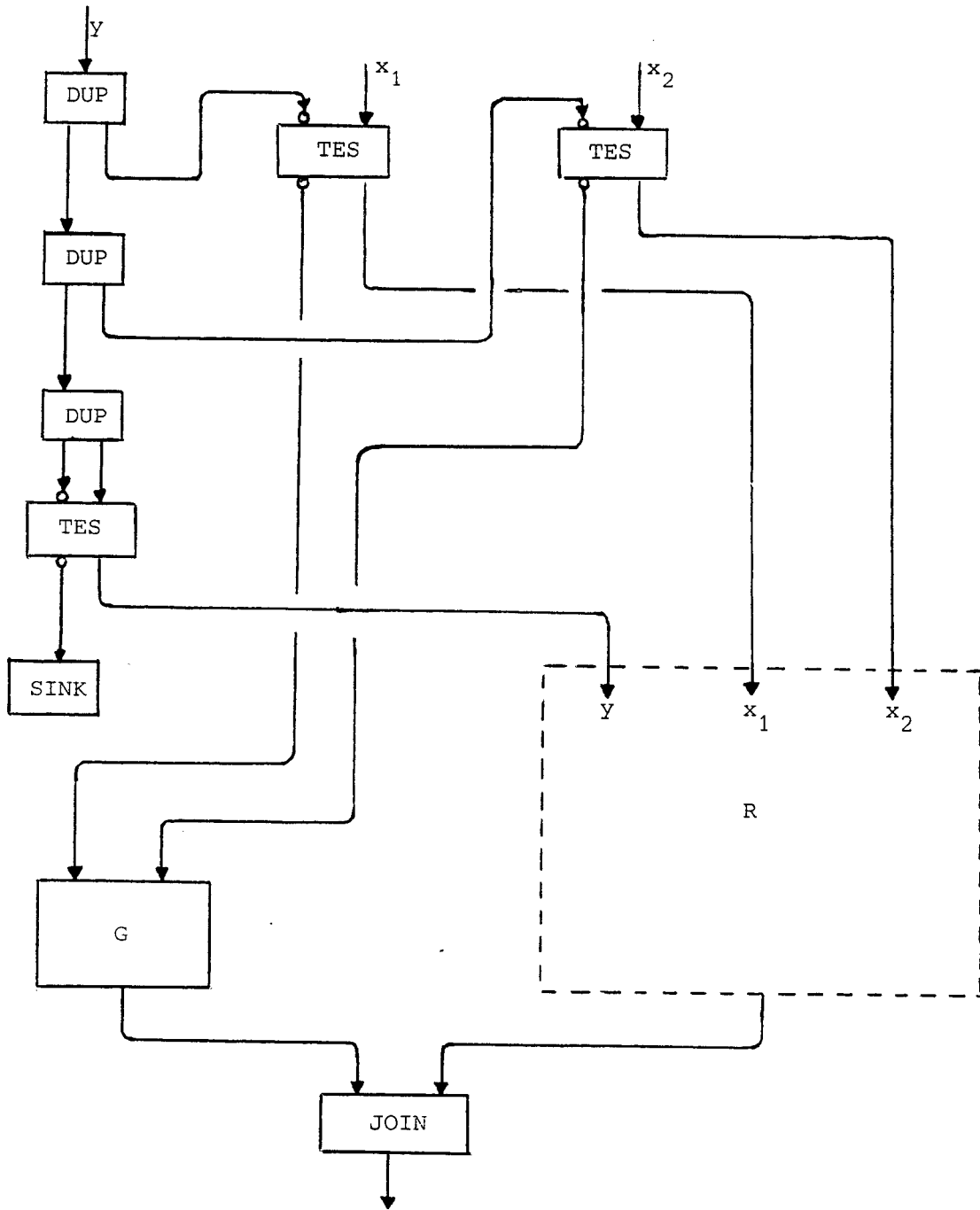


Figure 6. First design-step for N. (R still needs to be specified.)

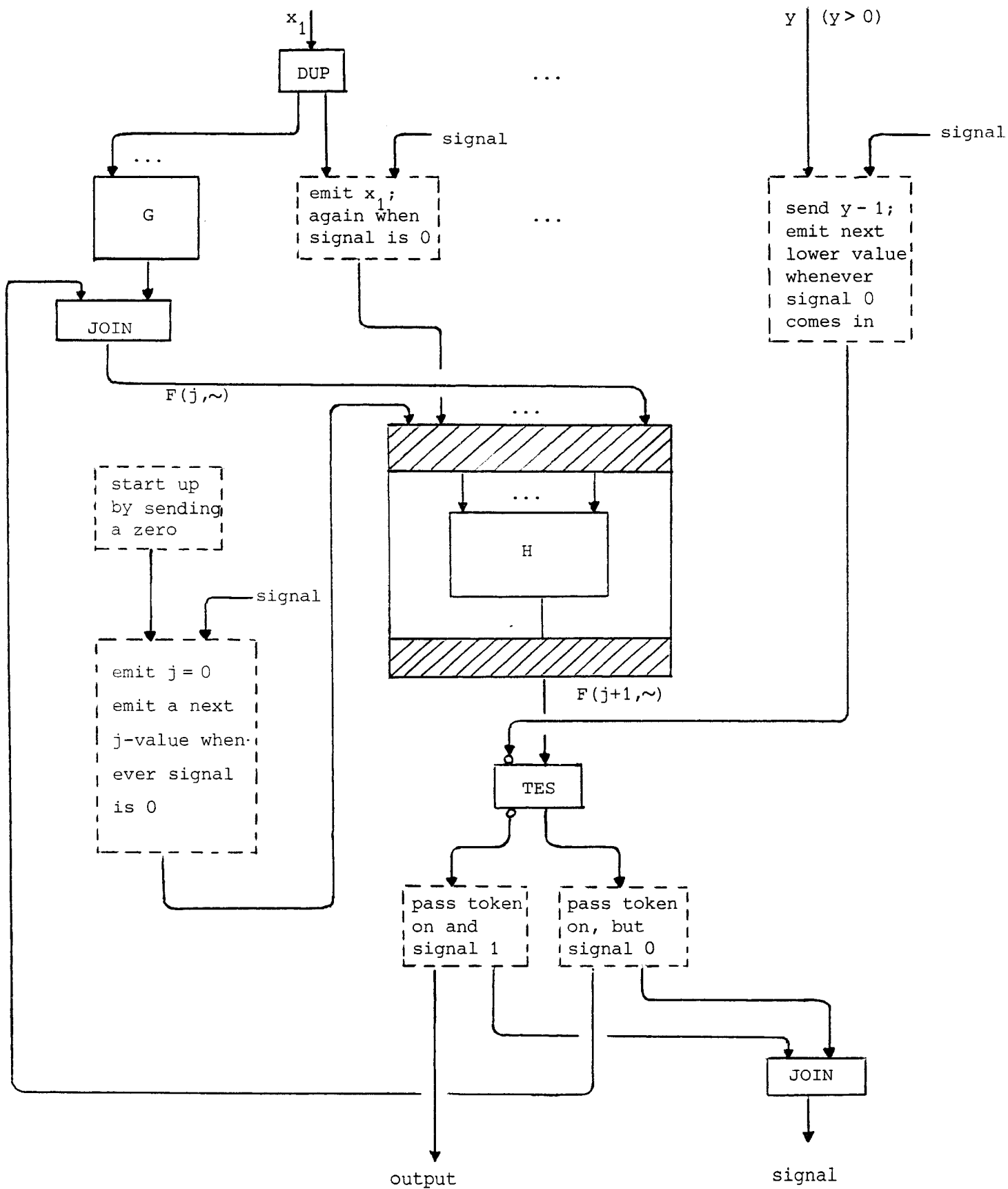
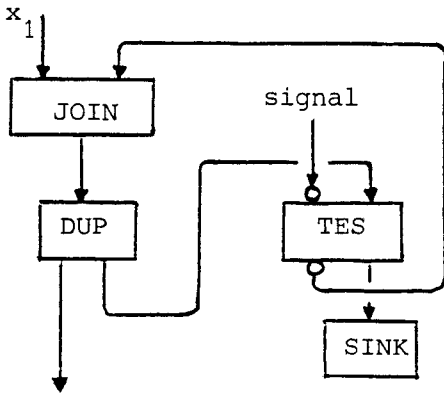
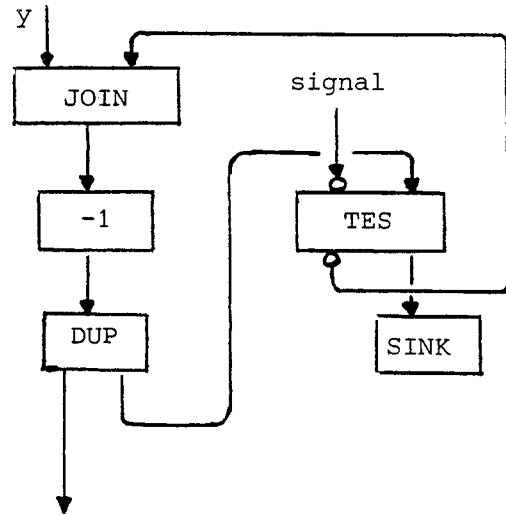


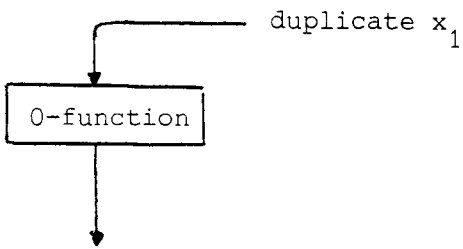
Figure 7. The R-graph



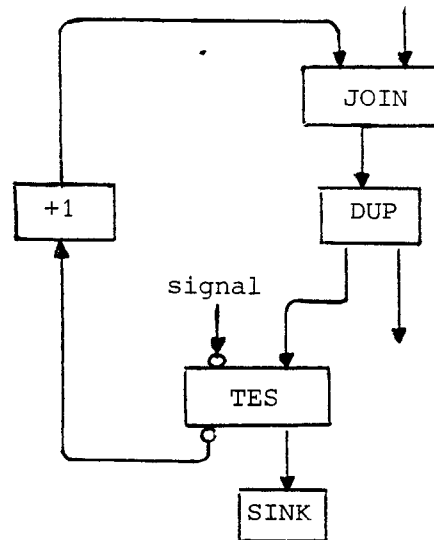
(a) emit x_1 ; again when signal is 0.



(b) send $y - 1$; send next lower value whenever signal 0 comes in.

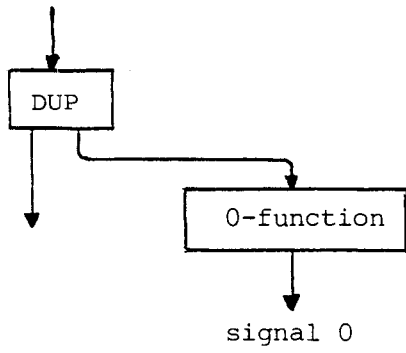


(c) start up by sending 0.

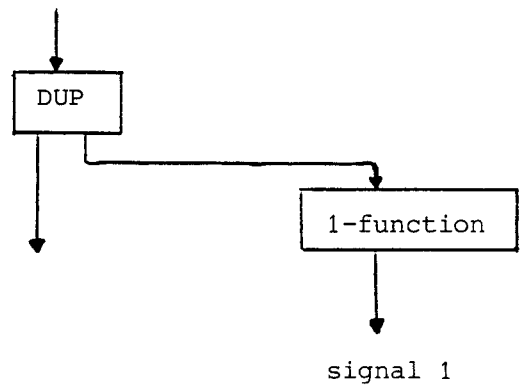


(d) emit $j = 0$; emit a next j -value whenever signal is 0.

Figure 8. Details of the R-graph.



(d) pass token on and
signal 0



(e) pass token on, but
signal 1

Figure 8. Details of the R-graph (cont'd).

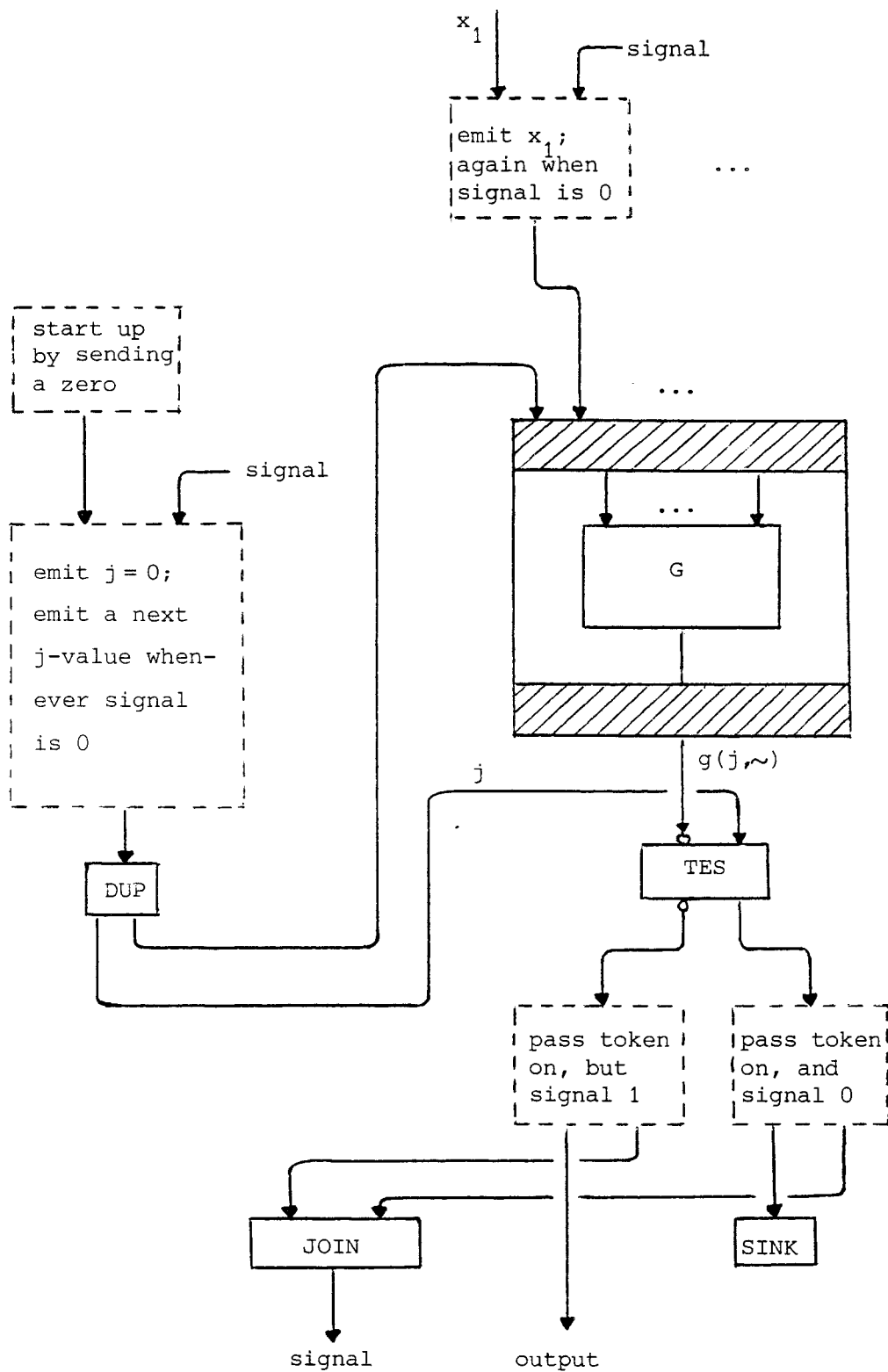


Figure 9. Global dataflow net for minimization.

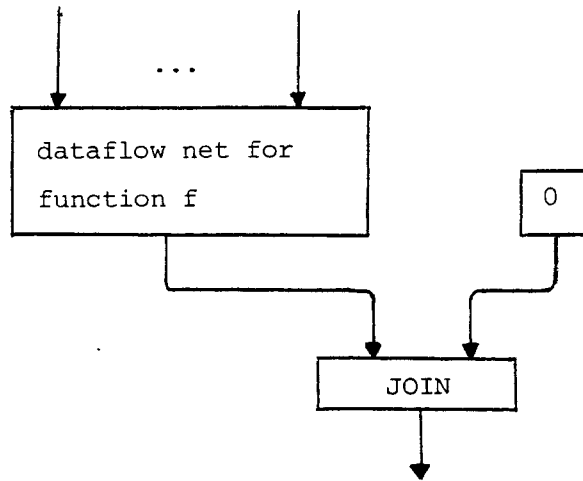


Figure 10. Well-formedness is undecidable.

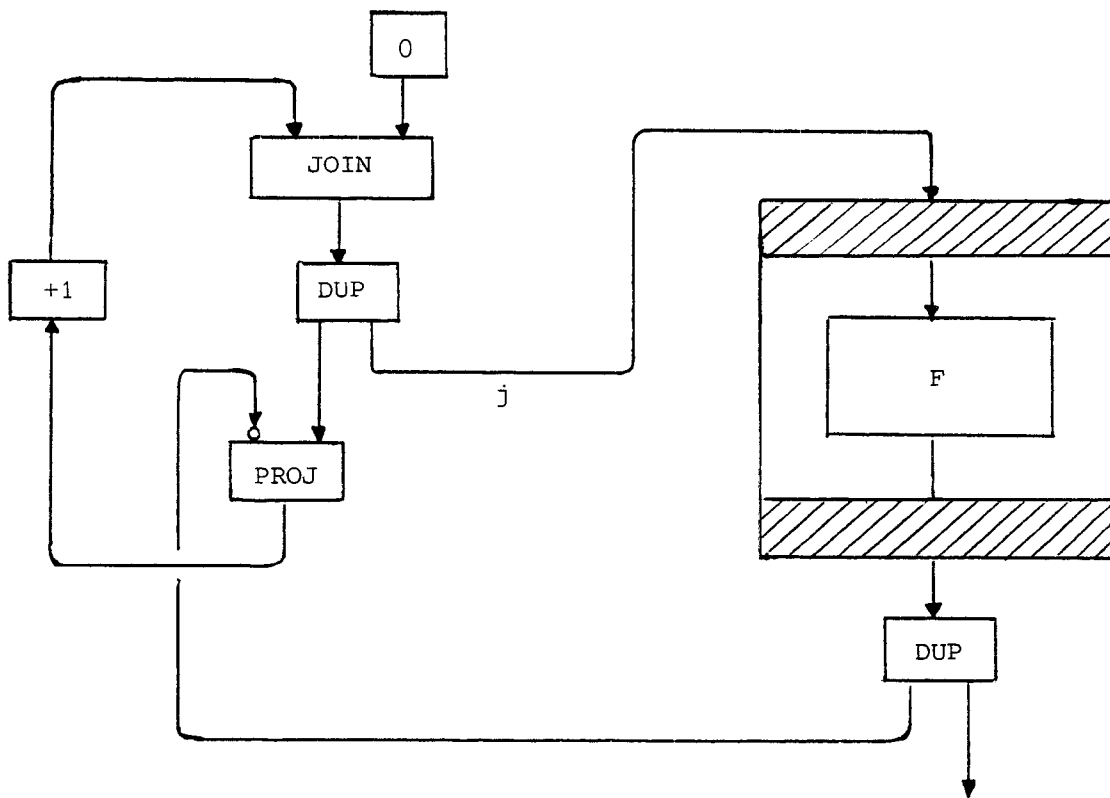


Figure 11. Generating a non-empty rec. enum. set.