An Alternative Approach to History Sensitive Computation in Dataflow Model

S. BANDYOPADHYAY*, S. GHOSH**, C. MAZUMDAR** and S. BHATTACHARYA**

Dataflow computation models generally discuss functional and a historic programs. This paper proposes a model, based on the concept of state variables, to tackle history sensitive computations in dataflow. A few history sensitive primitives are also proposed in which the concept of states are in-built. These primitives are not pure dataflow primitives in the sense that they have internal memory to preserve states between executions. This model is thus not free from side effects. However, an elegant modelling of some difficult history-sensitive computations can be derived from this approach.

1. Introduction

A number of applicative programming systems have been proposed in recent years to extract maximum parallelism in programs and to avoid Von Neumann bottleneck [1]. Most of these computation models generally discuss functional programs which are ahistoric in nature. However, history-sensitivity in computational models is practical and unavoidable in many situations like real-time problems, operating systems and DBMS. One of the formal attempts to solve history-sensitive problems is the Applicative State Transition system based on FFP proposed by Backus [1].

Of the other parallel programming systems, dataflow computation model is discussed in recent years [2, 3]. The functional and ahistoric nature of dataflow computation model exhibits a remarkable shortcoming in performing history-sensitive computation. In dataflow models history-sensitive problems can be represented by using the data type stream [4, 5] where the input history is preserved in the data type itself. Programs that manipulate streams can be written either in a recursive style [6] or in an iterative style [7]. Unfortunately, the handling of recursion is very inefficient in contemporary dataflow machines [8]; and the iterative schemas are difficult to prove and verify as the complexity increases with the order of dependences between input tokens.

This note discusses another approach to the handling of history-sensitivity in dataflow environment. This involves the fitting of history-sensitive computations in an automata-theoretic model, where a state-variable stores the required data connected with the input history. For the efficient application of this model, a few tiny history-sensitive primitives are also proposed. It is shown how these primitives may help a programmer in synthesizing history-sensitive programs in an elegant and compact manner.

*Dept. of Computer Sc. & Engg., Indian Institute of Technology, Bombay 400 076, India
**Dept. of Electronics, Jadavpur University, Calcutta 700 032, India

2. A General Model

In this section a general model of history-sensitive computation using two general functions with feedback is given. The functions themselves do not contain feedbacks and can be represented as acyclic dataflow graphs. Here, however, those functions are represented in FP [1] notations because it is more compact and simpler than the corresponding dataflow graph representation. The FP notations used by Backus is given in the appendix. It is to be remembered that FP, as proposed by Backus, neither support a data structure of arbitrary length like stream nor allow any lenient operator like *f* by to be used; all the operators are strict and have their arguments well defined when operating. However, even with these differences, the proposed representation scheme is useful even in the field of infinite structure. Subsequently, this representation scheme will be used to represent dataflow programs.

2.1 Automata Based Model

A history-sensitive computation is such that with a sequence of data objects as input, the current output depends not only on the current input, but also on the history of inputs.

Let us consider an example of continued sum of an input sequence which is described as:

Input sequence $= \{x_1, x_2, x_3, \ldots, x_i, \ldots\}$

Output sequence $= \{y_1, y_2, y_3, \ldots, y_i, \ldots\}$

such that $y_i = C + \sum_{k=1}^{i} x_k$.

It should be noted here that the relation between ahistoric and history-sensitive dataflow model is similar to the relation between combinational and sequential

switching circuits. It is in the context of sequential circuits in classical switching theory that the concept of state plays a vital role. The same applies to history-sensitive computation in dataflow model also and therefore an automata-theoretic model can be furnished as described below:

Definition:

A history sensitive dataflow model is a 6-tuple (S, I, O, f, g, c), where

S=a nonempty set of states,

I=a nonempty set of inputs,

O=a nonempty set of outputs,

g: $S \times I \rightarrow S$

f: $S \times I \rightarrow O$

$c \in S$ is the initial state.

A special input symbol \$ ($\$ \in I$) is used to terminate a sequence of arbitrary length. Thus, when a \$ input comes, the output becomes \$ and the state changes to initial state (c) to make the program ready for the next set of inputs. So in the model of history-sensitive computation, as shown in figure 1, the functions F and G not only contain f and g, but also respond to \$ input as explained above. They operate on the object ⟨state, input⟩.

Thus $F = \text{eq} \circ [2, \$] \rightarrow \bar{\$}; f$

and $G = \text{eq} \circ [2, \$] \rightarrow \bar{c}; g$

Any history-sensitive program modelled after this will be represented as HIST (f, g, c) where functions f and g have the list ⟨state, input⟩ as operand.

2.2 Examples

Using the above model, two history-sensitive problems are solved below.

With a sequence of inputs $\{x_1, x_2, x_3 \ldots\}$

the sequence of outputs for the computation of moving average of order 3 is shown in Table 1. The corresponding states required for the preservation of history are also tabulated there.

Formally, the problem can be defined as:

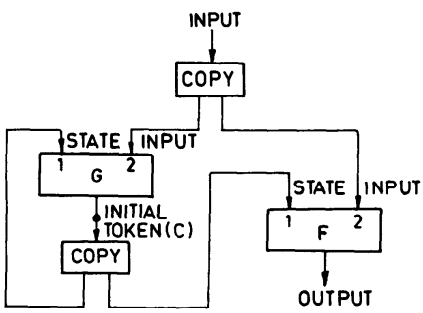


Fig. 1 Dataflow Model for History Sensitive Computation.

Input stream $= \{x_1, x_2, x_3, \ldots\}$

Output stream $= \{y_1, y_2, y_3, \ldots\}$

such that $y_1 = x_1/3$, $y_2 = (x_1 + x_2)/3$ and

for all $i >= 3$, $y_i = (x_i + x_{i-1} + x_{i-2})/3$

Clearly the output can be obtained by adding the input to the sum of the two components of the state and dividing the result by three. Now, as f and g is applied on the list ⟨state, input⟩, application of the selector function 1 (select the first element) on the list will give the state part and the selector function 2, the input part (see appendix).

So, here $f = (\text{div}\bar{3}) \circ + \circ [2, + \circ 1]$

where $(\text{div}\bar{3}) = \div \circ [\text{id}, \bar{3}]$.

Again obviously the new state can be obtained by composing the first element of the state to the input.

Hence $g = [2, 1 \circ 1]$.

The initial state is the list ⟨0, 0⟩.

So, the solution to the problem is

$$\text{MAV} = \text{HIST}\ (\text{div}\bar{3} \circ + \circ [2, + \circ 1], [2, 1 \circ 1], \langle 0, 0 \rangle)$$

Example 2: Another example of history-sensitivity is the selective transformation problem. It is defined as follows:

Input stream $= \{x_1, x_2, x_3, \ldots\}$

Output stream $= \{f_1(x_1), f_2(x_2), f_1(x_3), f_2(x_4), \ldots,\}$

where f_1, f_2 are any functions.

This leads to a solution where the output can be obtained by applying f_1 to input if the state is "true" and applying f_2 to input if the state is "false". The new state is complement of the old one. Initial state is "true" (T). Thus the solution is

$$\text{CONVERT} = \text{HIST}(1 \rightarrow f_1; f_2, \text{not} \circ 1, T).$$

Table 1 Trace of Example 1.

input	state	output
x_1	⟨0, 0⟩	$x_1/3$
x_2	⟨x_1, 0⟩	$(x_1+x_2)/3$
x_3	⟨x_2, x_1⟩	$(x_1+x_2+x_3)/3$
x_4	⟨x_3, x_2⟩	$(x_2+x_3+x_4)/3$
x_5	⟨x_4, x_3⟩	$(x_3+x_4+x_5)/3$

Table 2 Trace of Example 2.

input	state	output
x_1	T	f_1: x_1
x_2	F	f_2: x_2
x_3	T	f_1: x_3
x_4	F	f_2: x_4
.	.	.
.	.	.

2.3 Discussions

a. The chief advantage of this approach is the simplicity of the concept, which is in full conformity with the classical automata-based models for tackling history sensitivity as in sequential machines. No difficult-to-handle data structures like streams are necessary here. The computation model contains a global loop but no recursion which is difficult to handle in existing dataflow machines.

b. Once the trace of the problem is written down, the problem of determining f and g is as difficult or as simple as the problem of constructing a functional program from examples. Also, the solutions in the automata based model may sometimes turn out to be much simpler than the corresponding stream oriented solution.

c. The presence of global loop (see fig. 1) is in general deterrent to pipelining [9] and hence to parallelism. Moreover, an acyclic graph is easier to conceive. An interesting alternative here would be to split up a large history sensitive program into an acyclic network of a number of smaller programs, some of which may be history-sensitive, others functional. This also points to the necessity of devising history sensitive primitive building blocks to tackle general history sensitive problems in closed form for dataflow models. Such an approach is described in the next section.

d. Moreover, the state here is defined by the contents of arc II in figure 1. So the state is totally consumed by the functions f and g and a completely new state is generated by g. Thus this model, while allowing history-sensitive computations, does not have a complex and distributed state and does not allow modifications of small portions of state conceptually. This is a distinct advantage over von Neumann type machines [8].

The \$ symbol, if properly used, prevents one set of computation affecting the results of the next set.

3. History Sensitive Computation in the Closed Form

3.1 Motivation

This section shows how a few tiny history-sensitive primitives may help in the development of history-sensitive programs without any global loop or any explicit feedback. Analogy may again be drawn here with the synthesis of sequential circuits, where the designer freely makes use of history-sensitive blocks like flip-flops, shift registers, counters etc.

3.2 A Few History-Sensitive Primitives

To illustrate the concept, a few history sensitive primitive blocks are described below:

a. $SR_n(c)$: Shift Right Register of order n

For this block, the state S is a current list

$$\langle s_1, s_2, s_3, \ldots, s_n \rangle$$

With the application of an input i, the output is the state itself and the new state becomes

$$\langle i, s_1, s_2, \ldots, s_{n-1} \rangle$$

Def $SR_n(c) = \text{HIST}(1, \text{apndl} \circ [2, \text{tlrol}], c)$

In a similar manner, Shift Left register of order n ($SL_n(c)$) can be defined.

b. Tag n: Tag of the n^{th} order

This block generates the serial number of the input data tokens from 1 to n in the modulo-n field.

Thus, **Def** $\text{Tag}\ n = \text{HIST}(1, \text{eq} \circ [1, n] \rightarrow \bar{1}; \text{add}\bar{1} \circ 1, 1)$.

c. ACC (op, c): Accumulator

The block start with an initial state c and performs a primitive binary operation 'op', on its state and input pair $\langle s, i \rangle$ to generate the next state. The output is always the next state itself.

Thus **Def** $\text{ACC}(\text{op}, c) = \text{HIST}(\text{op}, \text{op}, c)$,

where 'op' is a primitive binary operation.)

3.3 Examples

Using the blocks mentioned in sec. 3.2, the previous examples are again programmed. The elegance of the solutions would be obvious.

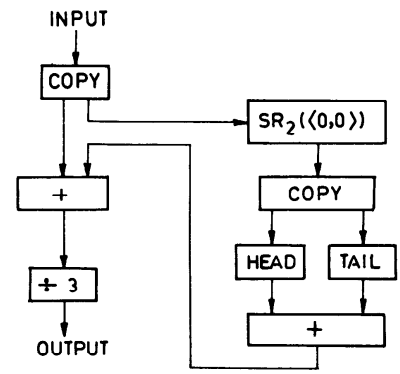
Example 1. The Moving Average Problem (MAV)

This has a solution shown in Fig. 2. SR_2 with initial state $\langle 0, 0 \rangle$ is used to preserve the previous two states.

If we are allowed to write the tiny history-sensitive blocks like SR2(0, 0) as functions, we can write the solution in the following form:

$$\text{MAV} = \text{div}\bar{3} \circ + \circ [\text{id}, + \circ \text{SR2}(0, 0)]$$

where $\text{div}\bar{3} = \div \circ [\text{id}, 3]$.



Note: HEAD, TAIL is used to extract the first and second component of the output list $\langle s_1, s_2 \rangle$ from SR_2

Fig. 2 Moving Average Problem.

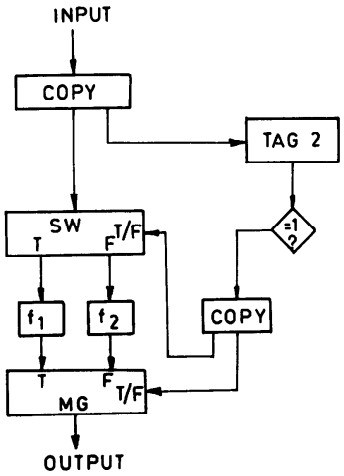


Fig. 3 Selective Transformation.

Example 2. Selective Transformation (CONVERT)

The solution is shown in Fig. 3.

This solution may be written as

$$\text{CONVERT} = \text{eq} \circ [\text{Tag}\bar{2}, 1] \rightarrow f1; f2.$$

3.4 Discussions

The distinct elegance of the solutions in section 3.3 over those of section 2.2 is mainly due to the fact that there is no explicit feedback in these solutions using tiny history-sensitive blocks. The history-sensitivity is achieved here by implicit feed-backs within the tiny history-sensitive blocks. This makes the programs easy to understand.

A more interesting study in this field may be that of designing a sufficient and powerful set of history sensitive primitives which can be used as the basic building blocks of any history-sensitive computation in data flow environment.

Appendix

Backus's Notations [1]:

Application $(f{:}x) =$ If f is a function and x is an object, $f{:}x$ is an application denoting the object $f(x)$.

Functions:

Selector Function $(s{:}x) \equiv$ If s is any integer, $n \geq s$ and $x = \langle x_1, x_2, \ldots, x_n \rangle$ then $s{:}x = x_s$; else undefined.

Tail $(tl{:}x) \equiv$ If $x = \langle x_1 \rangle$ then $tl{:}x =$ null; if $x = \langle x_1, x_2, \ldots, x_n \rangle$ and $n \geq 2$ then $tl{:}x = \langle x_2, \ldots, x_n \rangle$; else undefined.

Identity $(\text{id}{:}x) \equiv x$.

Equals $(\text{eq}{:}x) \equiv$ If $x = \langle y, z \rangle$ then if $y = z$ then $\text{eq}{:}x =$ true else false; else undefined.

Rotate left $(\text{rotl}{:}x) \equiv$ If $x = \phi$, then $\text{rotl}{:}x = \phi$; if $x = \langle x_1 \rangle$, then $\text{rotl}{:}x = \langle x_1 \rangle$; if $x = \langle x_1, x_2, \ldots, x_n \rangle$ and $n \geq 2$, $\text{rotl}{:}x = \langle x_2, \ldots, x_n, x_1 \rangle$; else undefined.

Rotate right $(\text{rotr}{:}x)$ is similar.

Append left $(\text{apndl}{:}x) \equiv$ If $x = \langle y, \phi \rangle$, then $\text{apndl}{:}x = \langle y \rangle$; if $x = \langle y, \langle z_1, \ldots, z_n \rangle \rangle$ then $\text{apndl}{:}x = \langle y, z_1, \ldots, z_n \rangle$; else undefined.

Append right $(\text{apndr}{:}x)$ is similar.

Right tail $(\text{tlr}{:}x) \equiv$ If $x = \langle x_1 \rangle$, then $\text{tlr}{:}x = \phi$; if $x = \langle x_1, \ldots, x_n \rangle$ and $n \geq 2$, $\text{tlr}{:}\ x = \langle x_1, \ldots, x_{n-1} \rangle$; else undefined.

Functional Forms:

Composition $(f \circ g) \equiv$ If f and g are any functions, then $f \circ g$ is a functional form such that for any object x, $(f \circ g){:}x = f(g(x))$.

Construction $([f_1, \ldots, f_n]) \equiv$ If $f_1, \ldots, f_n$ are any function, then $[f_1, \ldots, f_n]{:}x = \langle f_1(x), \ldots, f_n(x) \rangle$.

Condition $(P \rightarrow f; g) =$ If $p(x)$ is true, then $f(x)$ else $g(x)$ else undefined.

Constant $(\bar{x}) =$ If x is an object parameter, then for a defined y, $\bar{x}{:}y = x$, else undefined.

References

1. Backus, J. Can Programming Be Liberated From The Von Neumann Style? A Functional Style and Its Algebra Of Programs, *Communication ACM*, **21**, 8 (Aug. 1978).

2. Treleavan, P. C., Hopkins, R. P. and Brownbridge, D. R. Data Driven And Demand Driven Computer Architecture, *ACM Computing Surveys*, **14**, 1 (March 1982).

3. Dennis, J. B. First Version of A Dataflow Procedure Language, *Lecture Note in Computer Sc.*, **19**, Springer Verlag, New York.

4. Weng, K. An Abstract Implementation For A Generalized Dataflow Language, *Tech. Rep.*TR-228, *Lab. for Computer Sc., MIT*, Cambridge, Mass. (May 1979).

5. Arvind and Thomas, R. I-Structure: An Efficient Data Structure for Functional Languages, *Tech. Rep. (Revised)*, MIT/LCS/TM-178, *Lab. for Computer Sc., MIT*, Cambridge, Mass. (Octo. 1981).

6. Dennis, J. B. and Weng, K. An Abstract Implementation For Concurrent Computation with Streams, *Proc. of the* 1979 *Int's Conf. on Parallel Processing* (Aug. 1979).

7. Arvind, Gostelow, K. P. and Plouffe, W. An Asynchronous Programming Language And computing Machine, *Tech. Rep.*114a, *Dept. of Information and Computer Sc., Univ. of Californis, Irvine*, California (Dec. 1978).

8. Gajski, D. D., Padua, D. A., Kuck, D. J. and Kuhn, R. H. A Second Opinion On Dataflor Machines And Languages, *Computer*, **15**, 2 (Feb. 1982).

9. Davis, A. L. Data Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation For Distributed Control Systems, *Tech. Rep. UUCS*-78-108, *Dept. of Computer Sc., Univ. of Utah*, Utah (July 1978).

(Received June 26, 1987; revised September 19, 1988)