The University of Warwick

THEORY OF COMPUTATION

REPORT NO.28

AN EXTENSIONAL TREATMENT OF DATAFLOW DEADLOCK

BY

WILLIAM W. WADGE

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

April 1979

AN EXTENSIONAL TREATMENT OF DATAFLOW DEADLOCK

William W.Wadge
Computer Science Department
University of Warwick
Coventry CV4 7AL UNITED KINGDOM

Presented at the July 1979 Conference on Semantics of Concurrent Computations, Evian, France

Abstract

We discuss deadlock in reference to a simple equational dataflow language, and devise a test (the cycle sum test) which is applied to the dependency graph of a program. We use Kahn's extensional semantics of dataflow and give a purely extensional (non operational) proof that no program passing the cycle sum test can ever deadlock. The proof is based on the notions of size (length) and completeness in the domain of histories, and should extend to a much wider context.

0. Introduction

The question of termination has always been of fundamental importance in the theory of computation; in fact the single most important theoretical result is the unsolvability of the "halting problem". Of course, termination is of great practical importance as well. A conventional program which fails to terminate for some appropriate input is traditionally considered to be incorrect (or at least only partially correct, complete correctness being defined as partial correctness plus termination).

Recent developments, however, have to a large extent made obsolete the idea that proper programs all stop after some finite number of steps. Many programs now written are intended to perform continuously; for example an operating system, a traffic control system, or (the one we will use) a data flow network. For these programs there is still a notion of 'healthy' behaviour, but it is (superficially) the exact opposite of termination. If such a program halts at some stage, it is usually considered to be in error, the victim of a "crash" or of "deadlock". Obviously, conventional methods for proving termination of conventional programs need not be relevant.

The reason that the traditional notion of termination fails to extend to these more general contexts is that it is an operational notion. Fortunately, it appears that there is a static notion (which refers to data objects, and not computations) which corresponds to termination in simple cases, but also generalizes. We have in mind the notion of completeness.

A complete object (in a domain of data objects) is, roughly speaking, one which has no holes or gaps in it, one which cannot be further completed. In a standard 'flat' domain, all but the minimal elements are complete; in a domain of ordered pairs, the complete elements are those for which the components are complete; in a domain of finite trees, those in which all the leaves are complete; and in a domain of functions, those which are total, i.e. which yield a complete result when given a complete argument.

The distinction between complete and partial (or incomplete) objects could prove to be even more important than that between terminating and nonterminating computations. Complete objects are important because they are mathematically 'conventional', and the collection of complete objects in a domain usually enjoys conventional mathematical properties which are not true of the domain as a whole.

Completeness is, as we said, a static concept, but it also has operational significance. If our programming language has a denotational (mathematical, extensional) semantics, then we have a correspondence between programs and elements of an appropriate domain. From our limited experience, it seems that programs which behave in a healthy fashion will correspond to complete elements of the domain. For example, a Turing machine which always halts computes a total (i.e. complete) function.

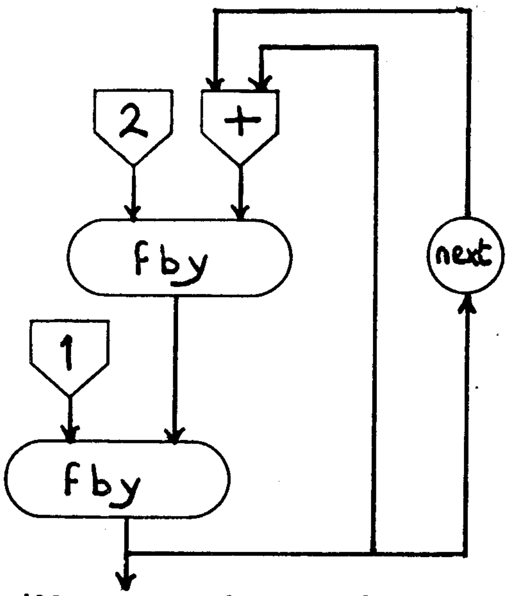
It might seem strange that we have defined so important a concept in such a vague and informal way. Unfortunately, there is at present little choice, for it seems that in general there is no way of determining which elements of an arbitrary domain deserve to be called complete.

In many domains the complete elements are exactly the maximal elements, but this need not always be the case; there may be incomplete elements which cannot be completed. In a domain of continuous functions, for example, there may be partial functions (like the as soon as functions of Lucid [2]) which have "essential" singularities. And even if completeness could be defined precisely in the context of standard domain theory (i.e. in terms of the relation of approximation), it would be of little help without methods for proving that objects defined in certain ways must be complete.

Fortunately there appears to be a simple way to extend the notion of domain to give a meaning to "completeness". The remedy is to adopt the methods of topology and introduce a quantitative measure of convergence, related to a metric. Rather than develop a general theory of such domains (at present only partially understood) we instead give an example of the power of the method by using it to justify the correctness of a simple and useful test for deadlock in a simple dataflow language.

1. Data Flow

The term "data flow" refers to a loosely defined operational concept in which computation is controlled by the flow of data through a network ([1], [3], [4], [7]). The model we will use is that of [7]. A data flow network is a directed graph, the arcs of which are communication channels down which data 'tokens' travel, and the nodes of which are processing stations. The diagram shows a simple data flow network which generates in order the sequence 1, 2, 3, 5, 8, ... of Fibonnacci numbers (our nets are continuously operating devices).



The example network illustrates the use of several important nodes.

The simplest are those like the one labelled "+" which correspond to ordinary operations on data items. The "+" node repeatedly awaits the arrival of tokens on its input lines; as soon as there are tokens on both lines, the two tokens are removed and a token representing their sum is sent down the output line. As with the other nodes, input on the different lines need not arrive simultaneously or even at the same rate, and tokens awaiting processing queue on the arcs. A special case of this kind of node are the 'constant' nodes with no input lines. The node labelled "2" simply generates an endless stream of tokens representing the number 2.

The remaining nodes do not process tokens but merely manipulate them. The "next" node discards the first token which arrives but passes the rest on.The "fby"("followed by") node awaits the first token to arrive on its left input, passes it on as its first output, but after that passes on whatever appears on its second input line. Any tokens which might arrive later on the first input line are discarded (but no input from the second line is discarded). The "fby" node allows arcs to be initialized, though possibly with values computed by other parts of the net. Both nodes have a two state internal memory.

These, of course, are not the only useful nodes. One particularly important one, which we will call the "upon" node (for want of a better name), acts as a valve or gate to slow up the rate of flow of its first argument. As long as 0's arrive down its second input line it sends on copies of the last token it read in from the first input line. When a 1 (representing "true") arrives it ingests the next token on the first line and sends on copies of it until another 1 arrives. For example, if tokens representing 3, 5, 7, 9, 11, ... arrive down the first input line, and tokens representing 0, 0, 1, 0, 1, 1, 0, 0, 1, ... arrive down the second input line, then tokens representing 3, 3, 3, 5, 5, 7, 9, 9, 9, 11, ... will be sent down the output line. This node has an internal 'storage location' capable of holding one data item.

Many other nodes have been proposed or are possible, but even with the few described here one can (as we shall see) write interesting programs.

2. The Extensional Semantics of Data Flow

The nodes just described all possess an extremely important property, namely functionality. A node is said to be functional iff the entire history of its output activity is completely determined by the entire histories of the activities of its input lines. Roughly speaking, this means that there are no random devices in the node itself, and that the rate of arrival of inputs effects only the rate of departure of outputs. A functional node can obviously take into account the order of arrival tokens on a particular line but not the relative order of arrival of tokens on different lines. The canonical example of a nonfunctional node is the 'race' or 'collector' node which passes on down its single output line whatever appears at either of its inputs (choosing at random if tokens are waiting on both lines).

Functionality is extremely important because it allows a simple extensional (i.e. mathematical or denotational) treatment of data flow. It allows us to use mathematical objects to characterise the role or function of arcs, nodes and graphs.

The mathematical object assigned to an arc is a history, namely a record of all the tokens which travelled down the arc. In a 'healthy' net the activity proceeds indefinitely, so that the history will be an infinite sequence of data items; but (as we shall see) it is possible that activity might cease at some finite stage, and so our domain of histories contains all finite and infinite sequences. This domain is a cpo under the subsequence ordering.

The mathematical object which we assign to a node is the function from histories to histories which describes the correspondence between the inputs and the outputs of the node.

Suppose now that we have a net all of whose nodes are functional. Each arc is the output of some node, so that the history corresponding to it is the result of applying the function corresponding to this node to the histories of its input arcs. The history of each arc is therefore defined by a simple equation, and so to a dataflow net there corresponds a set of equations, one for each arc in the net. If the net has cycles in it, this set of equations is recursive. Kahn indicated in [7], and Faustini has proved in [5], that the actual operational behaviour of the net is exactly described by the least fixed point (solution) of the equations.

3. The Equational Dataflow Language

The result just quoted is generally accepted as being very important, but often only as an accomplishment of descriptive semantics. In this perspective histories are used to describe the activity on arcs, functions describe the activities of nodes, and least fixed points of equations describe the activity of nets. From this point of view the result appears somewhat limited, because minor variations in the operational basis (e.g. nonfunctional nodes) cause the whole system to break down.

The real significance of the result emerges only when we reverse the point of view. Arcs and tokens should be seen as operational ways to realize histories; nodes, as implementations of history functions, and nets as devices for computing the solutions of equations. From this perspective many of the variations on pure data flow can be simply rejected as unsuitable for the purposes intended.

One very interesting feature of this new perspective is that data flow programs are not graphs but rather sets of equations. This equational language is quite easy to use and the programs are concise and often very elegant. Here is an equational version of the Fibonnacci program

```
F = 1 fby (2 fby F + G)
G = next F
```

and here is one which generates the stream of factorials

```
I = 1 fby I + 1
F = 1 fby F * next I
```

(fby appears as an infix operator with lowest possible precedence, so that e.g. "2 fby F + G" is the same as "2 fby (F + G)"). Notice that nested expressions are permitted on the right hand side. Subexpressions (like "next I") correspond to 'anonymous' arcs in the dataflow network, i.e. arcs which do not correspond to any program variable.

Here is a merge program

```
XX = X upon XX ≤ YY
YY = Y upon YY ≤ XX
 Z = if XX ≤ YY then XX else YY
```

which has two input (i.e. undefined) variables, "X" and "Y" ("upon", like "fby", has low precedence). Given any values for these variables, the least fixed point of the program gives us the corresponding value of "Z". It is not hard to see that if X and Y are increasing streams of natural numbers then Z is their ordered merge, without repetitions. For example, if X begins 3, 5, 7, 9, 10, ... and Y begins 2, 6, 7, 9, 12, ... then Z begins 2, 3, 5, 6, 7, 9, 10, ...

If we add to the above program the equations

```
S = 1 fby Z
X = 2 * S
Y = 3 * S
```

we have a new program without input variables which generates in order all numbers of the form 2^i3^j (S begins 1, 2, 3, 4, 6, 8, ...).

The following program

```
AA = A upon B < 2
 B = AA fby if B > 1 then B÷2 else next AA
 D = B mod 2
```

generates and concatenates the binary expansions of the numbers in A (so that if A begins 9, 6, 8, 3, ... then D begins 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, ...). If we add the equation

```
A = 1 fby 2 * A
```

D will be the sequence 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, ...

4. Circularity and Deadlock

All the example dataflow programs given so far are recursive, i.e. have variables which are defined directly or indirectly in terms of themselves. This is true in general of all but the simplest programs, whether they generate streams of data (like the Fibonnacci program) or just process them (like the merge program).

The equational programmer uses circularity to bring about repetition (this accounts for its importance) although it is far more general and powerful than simple iteration. In operational terms, the fact that a variable is defined in terms of itself means that the corresponding arc in the net is part of a cycle (loop) in the net. Data tokens travel around and around the loops in a net, usually being transformed in the process. These loops are the 'wheels' that keep the net moving.

Real wheels in real machines have, however, a tendency to seize up and stop, and the same is true of the cycles in a dataflow net. If a net has a cycle in it, it means that some node is directly or indirectly consuming its own output. The possibility therefore arises that the node might starve itself, i.e. might find itself in a situation in which it is waiting for itself to produce data. This is what is usually (although in more general contexts) called deadlock.

Deadlock in a dataflow net can in fact occur. In the simple example

```
G = 1 fby (2 fby 1 + next next G)
```

the net turns out two numbers (1 and 2) and then seizes up when the two "next" nodes 'gobble up' the two tokens. Nothing further is produced, and the "+" node spends eternity waiting for its own output.

Another example is the following

```
Y = 5 fby next X
X = 2 * Y upon P
```

which deadlocks almost immediately unless the first value of P is 0. If it is 0, the extra copy of the 10 token makes its way past the "next" node and enables the second multiplication.

On the other hand, it is certainly possible to write circular programs which never deadlock provided only that their inputs do not run dry. This is the case with <u>all</u> the example programs given earlier.

Obviously, deadlock does not occur just because a variable is defined in terms of itself. A study of a small number of examples soon reveals that what matters is <u>how</u> a variable depends on itself. For example, in the following very simple healthy program

```
I = 1 fby I + 1
```

the variable I depends on itself but in such a way that the <u>present</u> value of I (i.e. the one currently being computed) depends on at most the previously computed value. On the other hand, in the following deadlocking program

```
J = 1 + J
K = 2 * next K
```

the present values of J and K depend on the present and future values of J and K respectively.

What is clearly indicated is some sort of requirement which would ensure that the present value of any variable depends only on its previous values.

The first step in formulating such a requirement is to describe more exactly the way in which the outputs of the various nodes depend on their inputs. For example, if

A = B + C

then the value of A being computed depends only on the values of B and C just read in; the output of a "+" node neither leads nor lags the inputs, and the first three (say) values of A require the first three values of B and C. On the other hand, if

A = next D

then the output lags the input by one : the first <u>three</u> values of A require the first <u>four</u> values of D (or at least require that the node read these values in). Conversely, if

A = 3 fby B

then the output <u>leads</u> the second input : the first three values of A require only the first two values of B.

These lead/lag effects are clearly cumulative. If we have

A = 3 fby (5 fby B)

then A leads B by 2, so that the first three values of A require only the first value of B. The effects can also cancel each other: if

A = 3 fby next B

then in general the first n values of A require the first n values of B.

These observations suggest some sort of quantitative measure of this time displacement of dependencies. We therefore associate with each of the arguments of the different operations an integer which (informally speaking) measures how far the output leads the argument in question. The associated numbers are as follows:

(i) 0 is associated with each argument of the "+" node, and in general with each argument of a node computing an ordinary data operation;

(ii) 0 and 1 respectively are associated with the arguments of fby;

(iii) -1 is associated with the argument of next;

(iv) 0 and 1 respectively are associated with the arguments of upon.

When operations are composed, these numbers are added; to find the way in which the value of a whole expression can depend on the values of variables occurring in it,we consider the expression as a tree, trace a path from the root of the tree to the variable, and add up the numbers associated with the operations on the path. For example, if the expression is

(3 fby (next B + next C)) upon (next (P fby B))

then the path to "P" goes through the second argument of "upon" (+1), the argument of "next" (-1), and the first argument of "fby" (0). The sum of these numbers is 0, and so we conclude that in general the present value of the expression could depend on the present value of P. If a variable occurs more than once in an expression, we take the minimum of the path sums (which, in the case of "B" here, is 0).

It should not now be too hard to guess how we can use these numbers to assure ourselves that a variable in a data flow program is not defined in terms of its own present or future values. We look at the graph of the program and form path sums for all paths which start and end with the arc corresponding to the variable in question (i.e. all cycles containing that arc). If every such "cycle sum" is positive, the dependency of that variable on itself is healthy. To make sure that the whole program is healthy, we perform the test for every arc. Equivalently, we make sure that <u>every cycle in the graph of the program has a positive cycle sum</u>.

This is the <u>cycle sum test</u> and our claim is that every program passing the test is immune to deadlock (provided only that its inputs do not deadlock). All the example programs given in the earlier section pass the test. In the graph of the Fibonacci program, for example, there are two cycles, and their sums are 1 and 2.

5. Justification of the Cycle Sum Test

Deadlock is an operational concept and so it might be expected that we will now proceed to an operational proof of our claim. This is possible, and worth doing,but it would be missing the point of this paper, which is to illustrate the possibility and importance of an extensional (non operational) notion of completeness.

The connection between deadlock and completeness is actually quite easy to appreciate. Even in the absence of a precise definition of dataflow deadlock, it is evident that deadlock cannot be present in a net in which the activity on arcs(flow of tokens) goes on indefinitely. In terms of extensional concepts, unceasing activity corresponds to infinite elements in the domain of histories previously described,and we can certainly agree that these are exactly the elements of that domain which deserve to be called complete. The equivalence of the operational and extensional semantics of dataflow tells us that to prove a program deadlock free it is sufficient to prove that its least fixed point is complete (for every complete set of values for its input variables). (Kahn himself noticed this connection between deadlock and completeness).

Our goal therefore is to show that the least fixed point of a set of equations passing the cycle sum test must be complete. We can get a good idea of why this must be the case by examining almost the simplest healthy recursive program,

```
I = 1 fby I + 1
```

whose least fixed point <1,2,3,...> is complete. Let f be the function defined by the right hand side, i.e. let f(x) be 1 fby x+1 for any history x, so that the meaning of the program is the least fixed point of f. We know that this least fixed point is the limit (lub) of the sequence $\emptyset$, $f(\emptyset)$, $f(f(\emptyset))$, ... , $f^i(\emptyset)$, ... (where $\emptyset$ is the empty history). This sequence of histories begins

$$\begin{array}{l} \emptyset \\ \langle 1\rangle \\ \langle 1,2\rangle \\ \langle 1,2,3\rangle \end{array}$$

and it is easy to see why the limit must be complete - the terms of this sequence increase in length by one on each step. Furthermore, it is easy to see why these lengths increase - the function f increases length, i.e. the length of f(x) (x finite) is one plus the length of x. In fact it is evident that the least fixed point of any length increasing function must be infinite (complete).

Since we measure the dependency of "I" on itself as plus one, it would seem likely that there is some connection between length and the numbers we assigned to the arguments of operations. This is indeed the case. For example, we associated the numbers 0 and 1 with the arguments of upon, and simple calculations will show that (if x and p are finite) the length of x upon p is at least the minimum of the length of x and the length of p plus one. In general we will associate a sequence d of integers of length n with an n-ary operation g on histories whenever the length of $g(x_0,x_1,\ldots x_{n-1})$ is, for any $x_0\ldots x_{n-1}$, at least

$$\min_i(\text{length}(x_i) + d_i)$$

To make these ideas more precise, let H be the domain of histories, let $\hat{Z}$ be the natural numbers plus ∞ (with numerical ordering) and extend addition and subtraction to $\hat{Z}$ in the obvious way.

<u>Definition</u> For any positive integers n and m and any n×m matrix M with components in $\hat{Z}$.

B(M) is the set of all functions g from H^n to H^m such that

$$\text{length}(g(x))_j \geq \min_i (\text{length}(x_i) + M_{ij})$$

for any j and any x in H^n.

The components of the matrix M estimate the way in which the ith input of g effects the jth output. For example, if $g \in B(M)$ and $M_{3,5}$ is 2, then the first k values of the 5th output of g require at most the first k-2 values of the 3rd input.

This association between functions and matrices has the following important property: that composition corresponds to min/sum matrix product. If $g \in B(M)$ and $n \in B(N)$ and if the composition of g and h is defined, then their composition is in B(L) where

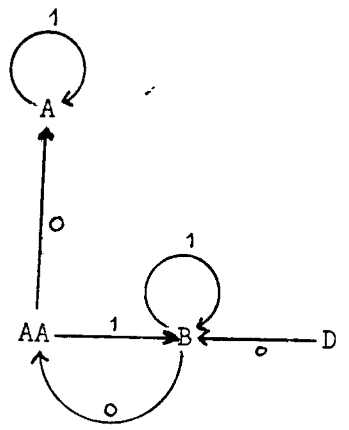
$$L_{ij} = \min_k \; M_{ik} + N_{kj} \; .$$

Now suppose that we have a general dataflow program (for simplicity without input variables) consisting of n equations defining n variables. From these equations we form the n×n matrix (with components in $\hat{Z}$) the i^{th} row of which consists of the numbers describing the way in which the expression on the right hand side of the equation defining the i^{th} variable depends on the other variables. For example, if the program

```
 A = 1 fby 2 * A
AA = A upon B < 2
 B = AA fby if B > 1 then B/2 else next AA
 D = B mod 2
```

then the matrix and its corresponding graph is

$$\begin{array}{c|cccc} & A & AA & B & D \\ \hline A & 1 & \infty & \infty & \infty \\ AA & 0 & \infty & 1 & \infty \\ B & \infty & 0 & 1 & \infty \\ D & \infty & \infty & 0 & \infty \end{array}$$



(notice that ∞ signifies no dependency)

This is the matrix of <u>direct</u> dependencies; the corresponding graph is not exactly the dataflow graph of the program, but it is apparent that both graphs pass or fail the cycle sum test together.

It is not hard to see that this matrix M can be associated with the function f from H^n to H^n defined by the equations of the program; in other words, f is in B(M). Thus for any x in H^n we have

$$\text{length}(f(x)) \geqslant M * \text{length}(x)$$

where length has been 'coerced' into a vector-to-vector operation and * is min/sum product. We must somehow put these facts together to conclude that the least fixed point of f is complete (that each component is complete).

If it happens that all the entries of the matrix M associated with f are positive, the result is guaranteed. The reason is that in these circumstances, f increases length in the following sense: the length of the shortest component of $f(x_0 \ldots x_{n-1})$ is always greater than the length of the shortest component of $\langle x_0 \ldots x_{n-1} \rangle$, and this condition easily implies that every component of the least fixed point of f is infinite.

Unfortunately it is almost never the case that all of the entries of the matrix M itself are positive, even when the program (like the example just given) satisfies the cycle sum test. In particular, if the program includes the equation

$$V_i = \text{next } V_j$$

then the i,j th component of M will be -1. The fact that the program passes the cycle sum test means only (in matrix terms) that the diagonal elements of the dependency matrix

$$\min_{1 \leq k \leq n} M^k$$

are positive. For example, the dependency matrix of the program just given is

$$\begin{array}{c|cccc} & A & AA & B & D \\ A & 1 & \infty & \infty & \infty \\ AA & 0 & 1 & 1 & \infty \\ B & 0 & 0 & 1 & \infty \\ D & \infty & \infty & 1 & \infty \end{array}$$

A little experiment shows that the problem is that it 'takes time' for the increases in the length of one component to effect the rest. This would suggest speeding up the process by iterating f, i.e. by considering an equation of the form

$$x = f^m(x)$$

for some large m. (The least fixed point of f^m is the same as that of f for any positive m). By what was said earlier we know that f^m is in $B(M^m)$, where exponentiation is with respect to the min/sum product. It is therefore enough to show that some power of M has all positive entries.

We mentioned that M represents the labelled dependency graph of the program. Since we are using the min/sum product, the i,j th entry of M^m must be the 'cost' (i.e. path sum) of the 'cheapest' path <u>of length m</u> from i to j in this graph. Since this graph has n nodes, any path through this graph must consist of a cycle free part of length not greater than n plus a number of cycles, each also of length not greater than n. There are only a finite number of cycle free paths, so that the cost of the cycle free part of any path must be not less than some number b independent of m. On the other hand, each cycle contributes at least +1 to the cost. Thus if m is greater than $(n+1)|b|$, every path of length m will have more than b cycles and will therefore have positive cost. In other words, all the entries of M^m must be positive, as desired.

This completes an informal outline of the purely extensional proof that every program passing the cycle sum test is deadlock free.

6. Application of the Test

The fact that a program passes the cycle sum test means first of all that it is free of deadlock, as we already indicated. The class of programs which pass the test is surprisingly large, and includes all those which correspond to simple iterative algorithms (written in Algol with for-loops). There are, however, quite sensible programs which fail the test but nevertheless do not deadlock, and it would be too restrictive to require that a legal dataflow program pass the test. For these more general programs other methods of proving completeness must be devised.

We also saw that the meaning (least solution) of a cycle sum test approved program is a complete element in the domain of histories. This means that the denotation of the variables are 'conventional' objects, and one very important consequence is the fact that we can use conventional mathematical rules in reasoning about them. For example, we can invoke the equations

$$(I + J) - J = I$$

$$\text{if true then } X \text{ else } Y = X$$

which may not be valid if X, Y, I and J are not complete.

Finally, the fact that a program passes the cycle sum test means that it has a unique solution (since its least solution is maximal) and this gives us a very powerful verification rule. For example, the equations

$$I = 1 \text{ fby } I + 1$$

$$J = 1 \text{ fby } J + 2*I + 1$$

pass the test; furthermore, some simple rules tell us that

$$I^2 = 1 \text{ fby } I^2 + 2*I + 1$$

so that I^2 satisfies the equation defining J. Since this equation has a unique solution, we conclude that $I^2 = J$. Note that this proof involved no induction.

7. Towards a General Notion of Completeness

The proof that a program passing the cycle sum test has a unique, complete solution was purely extensional, and clearly used only a few assumptions about the distinction between complete and partial elements of the domain of histories and the length function. It is quite plausible that the proof could be generalized to any domain equipped with similar notions of 'size' and completeness.

We have already mentioned that there seems to be no way, given an arbitrary domain, to single out a subset of complete elements; nor does there seem to be a general way to introduce a norm. Most interesting domains, however, are not arbitrary, they are not pulled out of hats. They are constructed from simple domains using domain operations (like cartesian product) and recursive definitions. For these domains it seems likely that we can define size and completeness in a natural way. In fact we have already seen a simple example, where we earlier (implicitly) defined the length of a tuple of histories to be the length of its shortest component, so that a tuple is complete iff all its components are.

One specially intriguing property of these (to some extent hypothetical) domains is that the collection of complete elements in a domain would seem to form a metric space, if we define the distance between two complete elements to be 2^{-s} where s is the size of the glb of the elements. In the case of natural number valued histories, the space of complete elements is the Baire space of classical descriptive set theory (see, for example, [8]).

It is not possible (as far as we know) to formulate the cycle sum theorem purely in terms of functions on an abstract metric space. But it is possible, however, if we use instead of a metric a dual notion which we call an "agreement": a function which assigns to any two points a nonnegative element of $\hat{Z}$ which measures how close together the points are, yielding ∞ if they coincide. This approach could allow a fixed point semantics for a large class of 'obviously terminating' recursive programs which would be mathematically 'conventional' in that it could completely avoid reference to partial objects and approximation.

8. Acknowledgements

The cycle sum test was first devised by the author in order to characterize a class of compilable Lucid programs; C. Hoffman used the test in [6]. M. Farah was probably the first to notice that Lucid programs passing the test had unique solutions (which might not be complete because Lucid's as soon as function is incomplete). The importance of the test in the study of dataflow became apparent in the course of supervising Faustini's researches, and the latter helped formulate the present version of the proof.

References

1. Arvind and Gostelow, Dataflow computer architecture: research and goals, technical report no. 113, Department of Information and Computer Science, University of California Irvine.

2. Ashcroft, E.A., and Wadge, W., Lucid, a nonprocedural language with iteration, CACM 20, no. 7, pp 519-526.

3. Davis, A.L., The architecture of DDM-1: a recursively structured data driven machine, report UUCS-77-113, Department of Computer Science, University of Utah.

4. Dennis, J.B., First version of a dataflow procedure language, MAC TM 61, MIT.

5. Faustini, A.A., The equivalence of the operational and extensional semantics of pure dataflow, Ph.D. (in preparation), University of Warwick, Coventry UK.

6. Hoffman, C.M., Design and correctness of a compiler for a nonprocedural language, Acta Informatica 9, pp 217-241 (1978).

7. Kahn, G., The semantics of a simple language for parallel processing, IFIPS 74.

8. Kuratowski, K., Topologie (I), Warsaw (1958).