The University of Warwick

THEORY OF COMPUTATION

REPORT NO.38

AN OPERATIONAL SEMANTICS FOR PURE DATAFLOW

BY

A. A. FAUSTINI

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

June 1981

An operational semantics for pure dataflow

by

A.A.Faustini

Department of Computer Science

University of Warwick

Coventry UK

ABSTRACT

We prove the equivalence between an operational and an extensional semantics for pure dataflow.

The term pure dataflow refers to dataflow nets in which the nodes are functional (i.e. the output history is a function of the input history only) and the arcs are unbounded fifo queues.

Gilles Kahn gave a method for the representation of a pure dataflow net as a set of equations; one equation for each arc in the net. We present a complete proof that the operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to its associated set of equations. Our model is completely general since our nodes have the universality property, in that, for any continuous history function there exists a node that will compute it. Moreover since our nets are not built from a set of sequential primitive nodes the model is not in the communicating sequential processes framework. On the contrary our nets have the abstraction property in that any net can be collapsed into a node.

The above proof gives complementary ways of viewing pure dataflow nets, that is, as either sets of equations or as graphs. It moreover gives rise to an elegant equational dataflow language. Pure dataflow then takes on an important role since it is a correct implementation for such a functional programming language; nodes being implementations of continuous history functions; arcs and datons being implementations of histories; and nets being mechanisms for computing the solutions to sets of equations.

0. Introduction

Dataflow is a method of computation in which the behaviour of a network of processes is controlled by the flow of data through the network. A dataflow net is a directed graph whose nodes are autonomous computing stations and whose arcs are channels down which datons* flow.

Although we give a general operational semantics for dataflow, we are mainly interested in dataflow nets with the following properties:-

i) Every node in the net is functional, i.e. the entire history of its outputs is determined only by the entire histories of its input (a history being the finite or infinite sequence of datons to have passed along an arc).

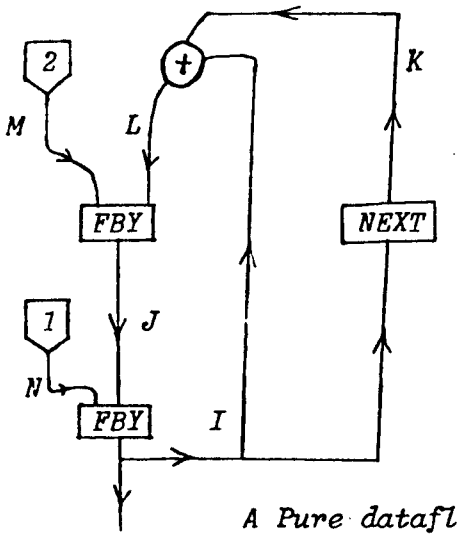
ii) Every arc is capable of storing datons in an unbounded fifo queue.

Nets with the above properties are called pure dataflow nets, so called because of their simple properties.

Gilles Kahn in [3] was the first to describe pure data flow and to point out that a pure dataflow net can be represented by a set of equations; one equation for each arc in the net. The diagram shows a pure dataflow net and its representation as a set of equations. Moreover Kahn realised that the operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to the net's equations. Although Kahn was the first to state this principle (which we call the Kahn principle) he did not prove it.

---

* We refer to an atomic unit of data as a daton. Throughout this paper we consider, without loss of generality, only datons of type natural number.



$I = N \text{ FBY } J$
$J = M \text{ FBY } L$
$K = \text{NEXT } I$
$L = K + I$
$M = 2$
$N = 1$

A Pure dataflow net and its representation as a set of equations.

We present a precise operational semantics for pure dataflow, which is then used to give a complete proof that the operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to the set of equations representing the net.

The Kahn principle has many important consequences, not the least of which is the fact that we can use sets of equations (like those above) as a dataflow programming language. This equational language, which is very close to the language Lucid, is concise, elegant and above all easy to reason about because it is simply mathematics as it stands.

From this point of view we do not consider a set of equations as representing (describing) pure dataflow nets; rather, we see pure dataflow nets as implementations of equational programs. It follows that we see pure dataflow nodes as implementations of history functions. From this point of view it does not matter how bizarre an implementation may be, if it computes a history function it is a pure dataflow node. For various reasons others who have studied pure dataflow ([0],[1],[3]) have restricted themselves to sequential nodes. We

argue that a complete operational semantics for pure dataflow should be able to describe any node that computes a history function. An example of a non-sequential node that computes a history function is the "parallel or" node, which we define more fully in a later section.

We present a completely general operational semantics in which we can formally describe any pure dataflow net. We are therefore able to give a complete proof of the Kahn principle.

1. Nodes as Automata

We think of nodes as a continuously operating processing stations ("black boxes"), with datons being fed in, one by one, through input arcs and datons being output, one by one, through output arcs. Although a simple node usually produces output at the same rate at which it consumes input, a more complex node may produce output at a different rate, perhaps faster or slower than the rate at which it consumes input.

With each node we associate an internal state which may change as the node moves from one stage in its computation to the next. When a node is first "activated", it moves automatically into a known initial state. Thereafter it may move to other internal states depending upon what the node is to compute. We can think of the internal state of a node as having two distinct functions.

One function is as a "marker", that is it marks the current step in the algorithm specifying the nodes behaviour. The initial state is a "marker" to the first step in such an algorithm. Consider a node with two internal states, one input arc and one output arc. In its first state (the initial state) it consumes the first daton to arrive on its input arc and produces no output. When that

step in the algorithm is completed it moves to its second internal state. In this state it outputs a copy of the next daton to be input. Since we require the node to repeat this last step ad infinitum, we force the node into the same internal state. The node we have just described is an implementation of the simple history function 'next'. (i.e. let α be a finite or countably infinite sequence of natural numbers, then next(α) is the sequence α with its first component removed.)

The second use of internal state is as memory, that is, to remember previous inputs. To produce an output, a node may require access to any of the previous inputs and, therefore, a possibly unbounded amount of internal memory may be used. An example of a node that uses internal state as memory is the node that produces, through its one output arc, the running total of the datons it has consumed through its one input arc. In this example we think of the initial state as initialising the running total to zero. When the first daton arrives, its value will be some natural number and since there are countably many natural numbers the node must have countably many internal states in order to remember the new running total. Since the sum of two natural numbers is again a natural number, a countably infinite number of internal states means that it is possible to record, using some internal state, any possible running total.

Although we can think of internal memory as having two distinct functions this does not mean that nodes need separate internal states for each of these functions. On the contrary, nodes may encode both of these functions within a single internal state.

A node is connected to its outside world through input and output arcs. It is along these arcs that datons flow into a node, and after some internal

computation flow out of the node. Consider two nodes in which the output arc of one node is the input arc of the other. If the source node produces datons at a faster rate than that at which the destination node consumes them, then the arc connecting these two nodes must be able to store the surplus datons in the order in which they arrived. For this reason arcs are taken to be unbounded fifo queues.

With each of a nodes input arcs we associate a one place input buffer. This buffer is empty if the input queue is empty, otherwise it holds the daton at the head of the input queue. A node is able to remove a daton from an input arc by erasing the contents of the corresponding input buffer.

The contents of each input buffer together with the internal state give a snapshot description of a node. This snapshot is called the "cause" of computation. With every possible "cause" we associate some "effect". An "effect" may be to erase some or all of the nodes input buffer; it may be to change internal state or it may be to ~~input~~ output a daton on some or all of the output arcs or a combination of these 3 activities.

For example, consider the node that computes the running total of its inputs. At some point in this node's computation a snapshot may reveal that it is in state q_{27} (the subscript denoting the running total so far). If a daton whose value is 3 is in the input buffer this will "cause" the following "effect": the 3 would be erased from the input buffer; the node would move to the new internal state q_{30} and a 30 would be output. We can see from this example how "causes" and "effects" are paired. We call such a pair a transition. The only property we require of transitions is that for every "cause" there is at least one possible "effect". Since a "cause" may have more than one "effect" associated with it, our nodes may be non-deterministic.

If one or more of the input buffers associated with a "cause" is empty, then it is still possible to associate an "effect" with that "cause". In some cases the "effect" may be to do nothing, which we call busy waiting. On the other hand the "effect" may be to cause some activity, this is called computing on empty buffers. It is possible for sequential nodes to compute when some of their buffers are empty, but only if they completely ignore the contents of these buffers. Using Kahn's Wait primitive, for example, it is possible to wait for the appearance of a daton down the first arc and output it when it arrives even if the second buffer is empty. But when a Wait is invoked, the node must do just that and has no way of knowing whether or not anything has arrived in the other buffer.

The more general nodes, however, are capable of performing other activities (such as output) while waiting for input on certain arcs - in other words, they are essentially able to do more than one thing at the same time. A very simple example of such a node is the 'double identity' node. This node has two inputs and two outputs and echos the first input on the first output, and the second input on the second output. Such a node cannot be sequential because it cannot allow both outputs to 'run dry' when only one of the inputs does so.

The following formal definition of a node is based on the informal ideas presented above. A node can be defined if: we know the number of input and output arcs; the initial internal state; the set of all possible internal states, and for every possible "cause" we have at least one "effect".

Definition *A* *node* *is a sequence* $\langle Q, q, n, m, T \rangle$

where

Q *is a finite or countably infinite set with* $nil \notin Q$ *(the set of all possible internal states)*

$q \in Q$ *(the initial internal state)*

$n, m \in \omega$ *(the number of input ports and output ports respectively)*

$T \subseteq (B^n \times Q) \times (E^n \times \hat{Q} \times B^m)$
(the transition relation)

such that
for any $c \in (B^n \times Q)$ *there exists* $e \in (E^n \times \hat{Q} \times B^m)$: $\langle c, e \rangle \in T$

where

$B = \omega \cup \{nil\}$

$\hat{Q} = Q \cup \{nil\}$

$E = \{tt, nil\}$

Given a node with n input arcs and m output arcs if we were to draw the node as a black box then we would number the arcs from the left (e.g. the left most input arc would be 0 and the right most would be n-1).

To clarify the above definition we examine in more detail the transition relation for a general node:

$T \subseteq (B^n \times Q) \times (E^n \times \hat{Q} \times B^m)$

The bracketing in the above expression delimits the "cause" and "effect" components described earlier. The "cause" component is:

$(B^n \times Q)$

A "cause" is of the form

$$\langle b_0, \ldots, b_{n-1}, q \rangle$$

where for any $i \in n$
b_i is the contents of input buffer i

$$b_i = \begin{cases} nil & \text{meaning the i'th buffer is empty} \\ x & \text{meaning the i'th buffer contains the natural number } x \end{cases}$$

The "effect" component is:

$$(E^n \times \hat{Q} \times B^m)$$

An "effect" is of the form

$$\langle e_0, \ldots, e_{n-1}, z, b_0, \ldots, b_{m-1} \rangle$$

where for any $i \in n$
e_i denotes what will happen to the contents of buffer i

$$e_i = \begin{cases} nil & \text{do nothing to the i'th buffer} \\ tt & \text{erase the contents of the i'th buffer} \end{cases}$$

z denotes what will happen to the internal state

$$z = \begin{cases} nil & \text{remain in the same internal state} \\ q & \text{move to internal state } q \end{cases}$$

for any $j \in m$
b_j denotes what will be output on arc j

$$b_j = \begin{cases} nil & \text{output nothing on the j'th output arc} \\ y & \text{output the natural number } y \text{ on the j'th output arc} \end{cases}$$

We now present a few examples to illustrate the definition.

1. We present a formal definition of the node with one input arc and one output arc that computes as output the running total of the inputs.

$\langle Q, q_0, 1, 1, T \rangle$

where

$Q = \{q_i \mid i \in \omega\}$

$T = \{\langle\langle nil, q_0\rangle, \langle nil, nil, nil\rangle\rangle,$

$\langle\langle 0, q_0\rangle, \langle tt, nil, 0\rangle\rangle,$

$\langle\langle 1, q_0\rangle, \langle tt, q_1, 1\rangle\rangle,$

$\langle\langle 2, q_0\rangle, \langle tt, q_2, 2\rangle\rangle,$

$\vdots \qquad \vdots$

$\langle\langle nil, q_{27}\rangle, \langle nil, nil, nil\rangle\rangle,$

$\langle\langle 0, q_{27}\rangle, \langle tt, nil, 27\rangle\rangle,$

$\langle\langle 1, q_{27}\rangle, \langle tt, q_{28}, 28\rangle\rangle,$

$\langle\langle 2, q_{27}\rangle, \langle tt, q_{29}, 29\rangle\rangle,$

$\vdots \qquad \vdots \qquad \}$

Let us examine two specific elements of T to see more clearly what is happening. Consider the transition

$\langle\langle nil, q_0\rangle, \langle nil, nil, nil\rangle\rangle$

The first component of this, $\langle nil, q_0\rangle$, *is the "cause", and the second,* $\langle nil, nil, nil\rangle$, *is the "effect". We would read the transition as follows, in the initial state with an empty input buffer do the following: do not erase the input buffer; do not change internal state and do not output anything (busy waiting). Consider the more active transition:*

$\langle\langle 2, q_{27}\rangle, \langle tt, q_{29}, 29\rangle\rangle$

We read this as: in state q_{27} *with a 2 on the input arc, do the following: erase the input buffer, move to internal state* q_{29} *, and output 29.*

Before we give another example we introduce the idea of a transition schema. This will give us a more concise way of defining the set of transitions. The following schema is equivalent to the set of transitions T given above.

for any $i,x \in \omega$

$\langle nil, q_i \rangle \rightarrow \langle nil, nil, nil \rangle$

$\langle 0, q_i \rangle \rightarrow \langle tt, nil, i \rangle$

$\langle x, q_i \rangle \rightarrow \langle tt, q_{i+x}, i+x \rangle$

Note the "cause" component appears to the left of the arrow ("->") and the "effect" component to the right.

2. As a second example, we present a node with two input arcs and one output arc. The node waits for a daton to arrive in either of the input buffers. When a daton arrives, it is consumed and a copy is output. If two datons arrive simultaneously, one at each buffer, then one of the buffers is chosen at random, the contents is consumed and a copy is output. This node is called the "unfair merge node".

$\langle Q, q, 2, 1, T \rangle$
where
$Q = \{q\}$

T is

for any $x,y \in \omega$

$\langle nil, nil, q \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle nil, y, q \rangle \rightarrow \langle nil, tt, nil, y \rangle$

$\langle x, nil, q \rangle \rightarrow \langle tt, nil, nil, x \rangle$

$\langle x, y, q \rangle \rightarrow \langle tt, nil, nil, x \rangle$
$\searrow \langle nil, tt, nil, y \rangle$

3. The third example is a node with two input arcs and one output arc. In a simple implementation the node waits for a daton to arrive in each input buffer, it then consumes both inputs and outputs the logical "or" of the

inputs. We assume that the only inputs are 1 and 0 denoting "true" and "false" respectively. We present a parallel implementation of the "or" function which we call "parallel or". In its initial state the node waits for a daton to arrive in either of the two input buffers. For the sake of argument we assume a "true" daton arrives on the left-hand input. The node will then, without waiting for the other input, erase the left-hand buffer, and move to an internal state which remembers that it has consumed a left-hand "true". It then outputs a "true" since no matter what the corresponding right-hand input is, it is still obliged to output a "true". If another "true" arrives in the left-hand buffer and still nothing has arrived on the right-hand side then it repeats the last step except that it moves to an internal state which remembers that it is two ahead on the left-hand input. The node can carry on like this indefinitely or until a "false" arrives on the left-hand input. Since it does not know ahead of time what the corresponding right-hand component will be it waits for right-hand input to catch up. (A similar algorithm can be used to implement "parallel and".)

$\langle Q, q, 2, 1, T \rangle$

where

$$Q = \{L_i \mid i \in \omega\} \cup \{R_i \mid i \in \omega\}$$

$$q = L_o = R_o$$

T is

for any $x, y \in \{T, F\}, i \in \omega$

$\langle nil, nil, q \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle T, nil, q \rangle \rightarrow \langle tt, nil, L_1, T \rangle$

$\langle F, nil, q \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle nil, T, q \rangle \rightarrow \langle nil, tt, R_1, T \rangle$

$\langle nil, F, q \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle x, y, q \rangle \rightarrow \langle tt, tt, nil, x\ y \rangle$

$\langle nil, nil, L_i \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle T, nil, L_i \rangle \rightarrow \langle tt, nil, L_{i+1}, T \rangle$

$\langle F, nil, L_i \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle nil, y, L_i \rangle \rightarrow \langle nil, tt, L_{i-1}, nil \rangle$

$\langle T, y, L_i \rangle \rightarrow \langle tt, tt, nil, T \rangle$

$\langle F, y, L_i \rangle \rightarrow \langle nil, tt, L_{i-1}, nil \rangle$

$\langle nil, nil, R_i \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle nil, T, R_i \rangle \rightarrow \langle nil, tt, R_{i+1}, T \rangle$

$\langle nil, F, R_i \rangle \rightarrow \langle nil, nil, nil, nil \rangle$

$\langle Y, nil, R_i \rangle \rightarrow \langle tt, nil, R_{i-1}, nil \rangle$

$\langle Y, T, R_i \rangle \rightarrow \langle tt, tt, nil, T \rangle$

$\langle Y, F, R_i \rangle \rightarrow \langle tt, nil, R_{i-1}, nil \rangle$

2. Nodes and Functionality

The nodes we have just defined, with the exception of the "unfair merge" node, possess an important property, namely functionality. A node is said to be functional iff the entire output history is determined by the entire input history, (i.e. the node computes a history function). Functionality is important since it allows us to reason about a node as a function rather than as a complex

set of transitions, in the same way that the Kahn principle allows us to reason about a net in terms of a simple set of equations rather than as a complex network of nodes and arcs.

Loosely speaking, there are two ways in which a node may fail to be functional:

i) if there is randomness within the node itself

ii) if the rate of arrival of datons effects more than the rate of departure of datons (i.e. the node is time sensitive)

The "unfair merge" node is a classic example of a non-functional node.

An obvious way of defining what it means for a node N to compute a history function f is to assume that when computation begins an infinite sequence of datons has already been placed on each of the node's input arcs. We simply require that there exist an (infinite) computation sequence which produces ("accumulates" might be a better word) f(α) (α the input) on the output arcs.

This "obvious way" of defining functionality turns out to be unnatural and incorrect, for a variety of reasons.

For one thing a node will never (in use) have an infinite sequence of datons on its input arcs. On the contrary, the input arcs are usually empty to begin with and even at some intermediate stages in the computation. Since we allow nodes to compute and even produce output while waiting for input, it is not enough to require that the node function properly (i.e. compute f) only when it is provided datons at a faster rate than it consumes them. For example, we can define a node which copies its input buffer when it is full but outputs zeros when the buffer is empty. This node would, according to the above definition, compute the identity function - but in actual practice it could use

its empty buffer transition with disastrous effect.

The second problem with the "obvious" definition proposed is that it requires only that f(α) possible as the output history, but not necessary. Since our nodes are non-deterministic, this distinction (between possible and necessary activity) is crucial. We can certainly define a node which outputs a random sequence of datons, and the node would, according to our hypothetical definition, compute every function!

Furthermore, we cannot repair this last problem by requiring that every sequence of transitions produces f(α) as output. This requirement is unfair (too strict) because it rules out any sort of control or direction of the activity of a node. Such control, however, is necessary because our nodes are non-deterministic devices capable of doing more than one thing (e.g. input and output) at the same time. If computation proceeds at random one vital activity may be neglected even though the computation as a whole never stops. We call such a situation "livelock" (the term is due to E. A. Ashcroft).

For example, we could design a node which computes the identity function but whose transitions code up two different internal activities. One activity is to build up an internal memory (queue) of inputs, an the other is to output stockpiled datons. The node is non-deterministic because each cause is associated with (possibly) two effects, one stockpiling and the other outputing. A computation sequence for which all but finitely many operations are stockpiling operations would be in livelock and would fail to produce the required output.

Of course we could avoid all these problems by restricting ourselves to deterministic and sequential nodes, but we reject this course for reasons

stated. Instead we formulate a more "dynamic" version of the obvious "static" definition given already.

The trouble with the "static" version is that it allows no "choice" in the sequence of transitions. The dynamic version must allow a node to be used in conjunction with a "fair" strategy for avoiding livelock. Strategies would be used to repeatedly choose the next transition to be performed, the choice being based on the previous history of the computation.

If we want to think in anthropomorphic terms, we can imagine a strategy being used by the controller of the node who is attempting to ensure that the node produces the correct output. The controller's strategy must work no matter how the input arrives from the environment of the node, i.e. no matter at what rate the input datons arrive. The fact that a node computes a function does not mean that the controller succeeds no matter what choices he makes; it means only that he has some strategy which ensures success in his battle with a "hostile" environment.

Our correct definition formalises this anthropomorphic view in terms of winning strategies for infinite games.

The idea that nodes require controlling strategies, in order to choose transitions, suggest the following infinite game.

Let N be a node

Let f be a history function

The infinite game G(f,N) is as follows

i) The game begins with the node in its initial state and all the arcs empty.

ii) The two players alternate in making moves, 'I' playing first.

iii) On each of his moves 'I' places a daton on some or all input arcs (possibly none).

iv) 'II' on each move chooses a transition.

v) 'II' wins iff he made an infinite sequence of moves producing $f(\alpha)$, α being the history produced by 'I' moves.

In this game a strategy for player 'II' is a monotonic function τ that takes a sequence of moves for 'I' and produces a sequence of (responses) moves for 'II'.

A winning strategy for 'II' is then a strategy τ such that if A is an infinite sequence of moves of 'I' that produce α then $\tilde{\tau}(A)$ is an infinite sequence of moves for 'II' that produce $f(\alpha)$. where $\tilde{\tau}(A)=\bigcup_{i\in\omega} \tau(A|i)$.

The use of infinite games allows the following definition of functionality:

Definition A node N is said to compute a history function f iff there exists a winning strategy for player 'II' in G(f,N) and any other strategy γ for 'II' is such that if A, an infinite sequence of moves for 'I', produces α then $\tilde{\gamma}(A)$ produces an initial segment of $f(\alpha)$.

2.1 RESULT Every history function computed by a node is continuous (in the sense of Kahn).

As nodes are seen as implementations of history functions it should be possible to implement any function as some node.

2.2 RESULT (the universality property)

> *Every continuous history function is the function computed by some node.*

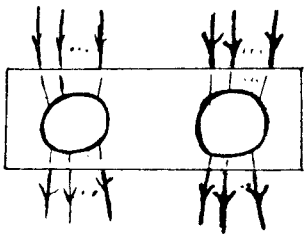
Hint to proof: It is possible to construct a node such that at the n'th step in the computation the node will have output $f(\alpha|n)|n$.

These definitions and results can also be applied to nets as well as nodes. Since a net also computes a continuous history function, Result 2.2 implies that any net can be collapsed into a node (the abstraction property). In other words, given any net (built up from functional nodes) there is a single functional node which can replace it in all pure dataflow contexts.

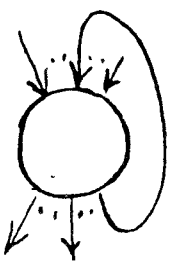
3. Nets and Functionality

There are two basic operations on nodes from which any legal net can be constructed, these operations are as follows:

i) Composition: the placing of two nodes together side by side.



ii) Iteration: joining the output arc of a node to an input arc of the same node.



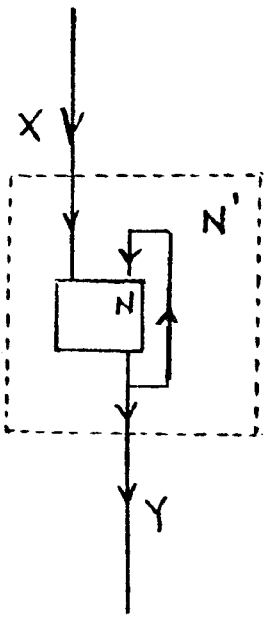
3.1 RESULT (the Kahn principle)

The operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to the net's equations.

Proof (outline)

Since it is not possible, in the space, to give a proof of the general case (see [2]) we prove the simplest non-trivial case.

Let N be a two input and one output node with the output bending back to the right-hand input arc.



Let τ be a winning strategy for the game G(f,N)

then there exist a winning strategy τ' for G(h',N') such that

$h' = \lambda x,\ \mu y\ f(x,y)$

N' is the one input one output node formed by encapsulating N and its right-hand input arc.

τ' is a strategy derived from τ using an auxiliary game in which τ is applied to X and its own output.

Since τ' is derived from τ, the first output of N' will be $f(X,\perp)|1$ (since N has no input to its right-hand arc). The second output of N' will be $f(X,f(X,\perp)|1)|2$ (since τ is playing against itself). If we continue the process we get the following

$Y|1 = f(X,\perp)|1$ (N' first output)

$Y|2 = f(X,f(X,\perp)|1)|2$ (N' second output)

$\vdots \qquad \vdots \qquad \vdots$

$Y|i+1 = f(X,Y|i)|i+1$ (N' i+1'th output)

$\vdots \qquad \vdots \qquad \vdots$

hence N' actually does compute

$\lambda x,\ \mu y\ f(x,y)$

4. Extensions

If result 3.1 (the Kahn principle) is seen as a result of descriptive semantics we would look for ways of extending the mathematical approach to handle a broader class of nodes and nets (i.e. not just pure dataflow). One such extension involves changing the basic domain of histories by introducing a special kind of daton called a "hiaton". A hiaton can be thought of as a unit of delay that (notionally) travels along with the ordinary datons and allows a node to produce something regularly even if it has no real output. Hiatonic streams code up timing information and it should be possible to use them to handle nodes and nets which are time sensitive.

On the other hand, if the operational semantics is seen as an implementation of a functional programming language (so that the Kahn principle states the correctness of the implementation) then we would look for ways to extend the language. The most obvious extension is to allow the user to include equations defining functions, including recursive equations. The implementation of such a language (which is similar to Structured Lucid) involves either dynamically growing nets or (notionally) infinite nets (but still pure

dataflow). The methods of this paper extend fairly easily to such nets and permit a proof of the correspondingly extended Kahn principle.

5. Acknowledgements

I would like to thank Bill Wadge for the time, effort and encouragement he has given to me as my research supervisor. I would also like to thank the other members of the Warwick dataflow group who also helped in various ways in the preparation of this paper.

References

[0] Arnold André

Semantique des processus communicants

Rapport de Recherche no.1 Juin 1979
Université de Poitiers

[1] Arvind and Gostelow, Kim P.

The Semantics of Asynchrony:
The relationships between two different interpreters of a programming language.

Technical report no.88
Department of Information and Computer Science
University of California, Irvine.

[2] Faustini A.A.

The equivalence between an operational and an extensional semantics for pure dataflow

Ph.D (in preparation)
University of Warwick, Coventry

[3] Kahn Gilles

The semantics of a simple language for parallel programming.

IFIPS 74

[4] Wadge W. W.

An extensional treatment of dataflow dealock.

July 1979 Conference on semantics of concurrent computations, Evian, France