Technical Memorandum UCB/ERL M97/3, Electronics Research Laboratory, Berkeley, CA 94720

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

January 15, 1997

A DENOTATIONAL SEMANTICS FOR DATAFLOW WITH FIRING

Edward A. Lee
EECS, University of California, Berkeley, CA, USA 94720.

Abstract

Formal semantics for the dataflow model of computation have centered around the version of dataflow known as Kahn process networks. These process networks, however, fail to capture an essential principle of dataflow, proposed by Dennis and used in almost all practical implementations of dataflow, that of an actor firing. An actor firing is an indivisible quantum of computation. A set of firing rules give preconditions for a firing, and the firing consumes tokens from the input streams and produces tokens on the output streams. These notions are missing from Kahn's model, and therefore have not been thoroughly studied in a formal setting. This paper bridges the gap, showing that sequences of firings define a continuous Kahn process as the least fixed point of an appropriately constructed functional. The firing rules are sets of prefixes with certain technical conditions to ensure determinacy. These conditions result in firing rules that are more general than the blocking reads of the Kahn-MacQueen implementation of Kahn process networks, and lead to a compositional dataflow model.

1. Introduction

Three distinct variants of the dataflow model of computation have emerged in the literature, Kahn process networks [8], Dennis dataflow [7], and dataflow synchronous languages [3]. The first two are closely related, while the third is quite different. This paper deals only with the first two, which have one key important difference. In Dennis dataflow, a process consists of a sequence of atomic *firings* of actors. Although Dennis dataflow can be viewed as a special case of Kahn process networks [10], the notion of firings has been absent from formal semantic models, which are most developed for Kahn process networks and dataflow synchronous languages. This omission is problematic because although Kahn process networks in their general form have not found widespread use, Dennis dataflow has, experimentally in computer architecture [2] and in production in signal processing software (see [10] and the references therein).

This paper fills in this gap, showing that methods pioneered by Kahn extend naturally to Dennis dataflow, embracing the notion of firing. This is done by establishing the relationship between a firing function and the Kahn process made up of a sequence of such firings. A practical consequence of this analysis is a formal characterization of firing rules and firing functions that preserve determinacy.

The semantics given here of *dataflow with firing* (Dennis dataflow) is *denotational,* in the sense of Scott and Strachey [13], rather than operational, the usual semantics given. The denotational semantics is shown to be equivalent to a usual operational semantics, thus establishing full abstraction.

2. Review of Kahn Process Networks

In all dataflow models, processes communicate by sending *tokens*, atomic units of data, along unidirectional channels with one writer and one reader. Let V denote the alphabet of tokens and S the set of all sequences of tokens. A particular finite sequence is written $[v_1, v_2, \ldots, v_p]$, and an infinite sequence $[v_1, v_2, \ldots]$. The set of tuples of n such sequences is denoted S^n. A particular n-tuple of sequences is written $\mathbf{s} = (s_1, \ldots, s_n)$. A Kahn process is a mapping $F: S^m \to S^n$ from an m-tuple an n-tuple, with a key technical restriction. The mapping must be a continuous function (in a sense reviewed below). This restriction ensures that compositions of Kahn processes are determinate (in a sense also reviewed below).

2.1 COMPLETE PARTIAL ORDERS AND THE PREFIX ORDER

An *ordering relation* on the set *S* is a reflexive, transitive, antisymmetric relation "$\sqsubseteq$" on members of the set. *Reflexive* means that $s \sqsubseteq s$, *transitive* means that $s \sqsubseteq s'$ and $s' \sqsubseteq s''$ imply that $s \sqsubseteq s''$, and *antisymmetric* means that $s \sqsubseteq s'$ and $s' \sqsubseteq s$ imply $s = s'$, for all s, s', s'' in S. Of course, we can define a related irreflexive relation, denoted "$\sqsubset$", where $s \sqsubset s'$ if $s \sqsubseteq s'$ and $s \neq s'$. A set *S* with an ordering relationship is called an *ordered set*. If the ordering relationship is partial (there exist $s, s' \in S$ such that neither $s \sqsubseteq s'$ nor $s' \sqsubseteq s$, then *S* is called a *partially-ordered set* or *poset* [6].

Below, we use the symbol N to denote the set of natural numbers, $\{1, 2, \ldots\}$, the symbol N_0 to denote $N \cup \{0\}$, and the symbol N_∞ to denote $N_0 \cup \{\infty\}$. The ordered set ω is N_0 with the usual numeric ordering relation.

A particularly useful partial ordering relation is called the *prefix order*. In the prefix order, $s \sqsubseteq s'$ if s is a prefix of s'. This means simply that the first n tokens of s' are the same as the n tokens in s, in the same order, where $n \in N_\infty$ is the length of the sequence s. The empty signal, one with no tokens, is denoted λ, and is a prefix of every other signal.

A *chain* in S is a set $\{s_i; s_i \in S \text{ and } i \in N\}$, where N is the set of natural numbers and $s_i \sqsubseteq s_{i'}$

$\Leftrightarrow i \leq i'$. An *upper bound* of a subset $W \subseteq S$ is an element $w \in S$ where every element in W is a prefix of w. A *least upper bound* (*LUB*), written $\sqcup W$, is an upper bound that is a prefix of every other upper bound. A lower bound and greatest lower bound are defined similarly. The *bottom* element of a poset, if it exists, is a lower bound for the poset itself. A *complete partial order* (*CPO*) is a poset with a bottom element where every chain has a LUB. From a practical perspective, this usually implies that our set S of sequences must include the empty sequence λ and sequences with an infinite number of values. The poset S with the prefix order is a CPO.

These definitions are easy to generalize to S^n, the set of n-tuples of sequences. For $\mathbf{s} \in S^n$ and $\mathbf{s}' \in S^n$, $\mathbf{s} \sqsubseteq \mathbf{s}'$ if each corresponding element is a prefix, i.e. $s_i \sqsubseteq s'_i$ for each $1 \leq i \leq n$, where $\mathbf{s} = (s_1, ..., s_n)$. Following Birkhoff and Mac Lane [5], we define S^0 to be a set with a single element. With this definition, if S is a CPO, so is S^n for any $n \in N_0$. The tuple of empty signals is denoted Λ, and is a prefix of every other tuple of signals of the dimension.

Consider a function $F : S^m \to S^n$, where $m, n \in N_0$. We allow either m or n to be zero in order to model Kahn processes with no inputs or no outputs, respectively. For example, a function $F : S^0 \to S^1$ is a source of a sequence. Since it is a function, and S^0 has only one element, the output is always the same sequence.

2.2 MONOTONIC AND CONTINUOUS FUNCTIONS

A function $F : S^m \to S^n$ is *monotonic* if

$$s \sqsubseteq s' \Rightarrow F(s) \sqsubseteq F(s'). \tag{1}$$

Intuitively, this says that if an input sequence s is extended with additional tokens appended to the end to get s', then the output $F(s)$ can only be changed by extending it with additional tokens to get $F(s')$. I.e., giving additional inputs can only result in additional outputs. This is like an untimed notion of causality. Note that if $m = 0$ or $n = 0$ then the function is always monotonic.

A function $F: S^m \to S^n$ is *continuous* if for every chain $W \subset S^m$, $F(W)$ has a least upper bound $\sqcup F(W)$, and

$$F(\sqcup W) = \sqcup F(W). \tag{2}$$

The notation $F(W)$ denotes a set obtained by applying the function F to each element of W. Intuitively, this says that the response of the function to an infinite input sequence is the limit of its response to the finite approximations of this input. Note again that if $m = 0$ or $n = 0$ then the function is always continuous.

"Continuous" here is exactly the topological notion of continuity in a particular topology called the *Scott topology*. In this topology, the set of all sequences with a particular finite prefix is an open set. The union of any number of such open sets is also an open set, and the intersection of a finite number of such open sets is also an open set.

A continuous process is always monotonic. To see this, suppose $F: S^m \to S^n$ is continuous, and consider two signals s and s' in S^m where $s \sqsubseteq s'$. Define the increasing chain $W = \{s, s', s', s', \ldots\}$. Then $\sqcup W = s'$, so from continuity,

$$F(s') = F(\sqcup W) = \sqcup F(W) = \sqcup \{F(s), F(s')\}. \tag{3}$$

Therefore $F(s) \sqsubseteq F(s')$, so the process is monotonic.

Not all monotonic functions are continuous. Consider for example a system where the set of tokens is binary, $V = \{0, 1\}$, and

$$F(s) = \begin{cases} [0]; & \text{if } s \text{ is finite} \\ [0, 1]; & \text{otherwise} \end{cases} . \tag{4}$$

It is easy to show that this is monotonic but not continuous.

A concatenation of a finite sequence s' and another (possibly infinite) sequence s is written $s'.s$. An example of a continuous function that we will discuss further is the *unit delay.* It is defined by

$$D_v(s) = [v].s \tag{5}$$

where $v \in V$ is a token value. The effect of the delay is to insert an initial token with value v onto the head of a sequence. The term "delay" reflects the fact that a given token on the input appears on the output also, but one token later in the sequence.

2.3 COMPOSITIONS OF KAHN PROCESSES AND DETERMINACY

A *finite composition* of Kahn processes is a collection $\{s_1, s_2, \ldots, s_p\}$ of sequences and a collection $\{F_1, F_2, \ldots, F_q\}$ of continuous functions relating them, such that no sequence is the input or output of more than one function. Any sequence that is not the output of any of the functions is an input to the composition.

A composition is *determinate* if given the input sequences, all other sequences are determined. Obviously, a Kahn process by itself is determinate, since it is a functional mapping from input sequences to output sequences.

Some examples of finite compositions of Kahn processes are shown in figure 1. In each of these examples, given the component functions, it is obvious how to construct a function that maps the input sequences (those that are not outputs of any function) to the other sequences. Thus, each of these compositions is determinate. Moreover, if the component functions are continuous (or monotonic), then the composite functions are also continuous (or monotonic).

Feedback compositions of Kahn processes may or may not be determinate. Consider for example the identity function, $I(s) = s$. This function is obviously continuous. Suppose we create a very simple composition of the identity process by feeding back the output to the input, letting $F = I$ in figure 2. There are no inputs to the composition, the composition is determinate only if the sequence s is determined. But any sequence s satisfies the composition, so it is not determined.

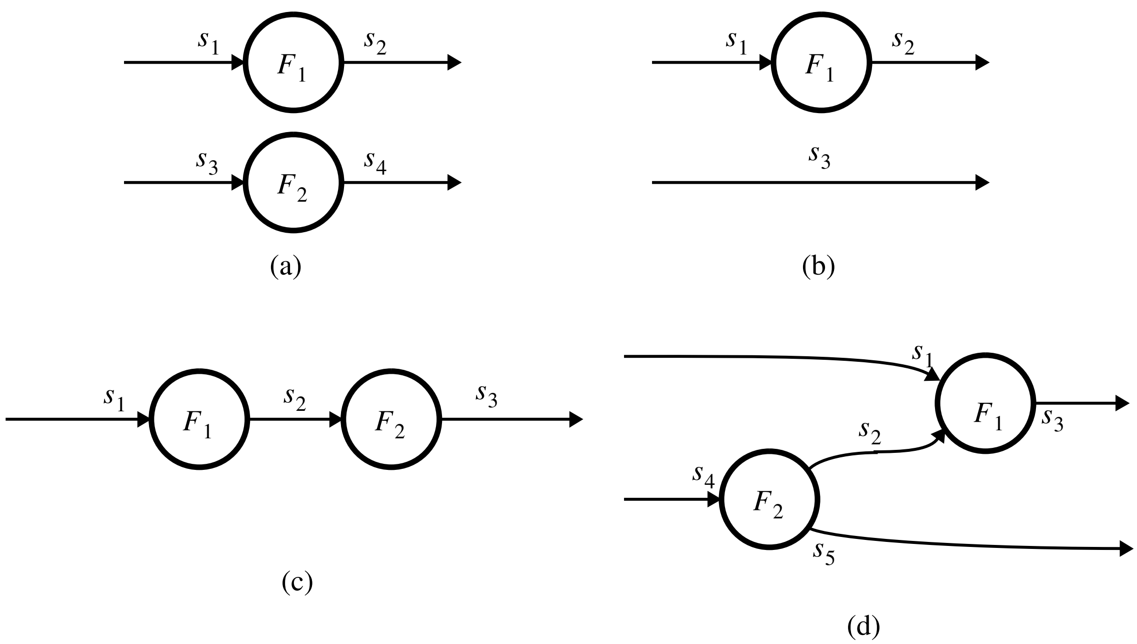

FIGURE 1. Examples of composition of processes.

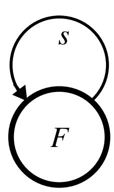

FIGURE 2. Feedback (a directed self-loop).

2.4 LEAST FIXED POINT SEMANTICS

There is an alternative interpretation due to Kahn [8] of a feedback composition that makes the example in figure 2 determinate. Under this interpretation, any composition of continuous processes is determinate. Moreover, this interpretation is consistent with execution policies typically used for such systems (their operational semantics), and hence is an entirely reasonable denotational semantics for the composition. This interpretation is the *least-fixed-point* semantics.

A well-known fixed point theorem states that a continuous function $F : X \to X$ in a CPO X has a least fixed point x, $F(x) = x$ (see [6], page 89). By "least fixed point" we mean that for any y such that $F(y) = y$, $x \sqsubseteq y$. Moreover, the theorem gives us a constructive way to find the least fixed point. Putting it into our context, suppose we have a continuous function $F : S^n \to S^n$. Then define the sequence of sequences

$$\mathbf{s}_0 = \Lambda, \mathbf{s}_1 = F(s_0), \mathbf{s}_2 = F(s_1), \ldots \qquad (6)$$

Since F is monotonic and the tuple of empty sequences Λ is a prefix of all other tuples of sequences, this sequence is a chain. Since S^n is a CPO, this chain has a LUB. The fixed-point theorem tells us that this LUB is the least fixed point of F.

This theorem is very similar to the so-called *Knaster-Tarski fixed point theorem*, which applies to complete lattices rather than CPOs [6]. For this reason, this approach to semantics is sometimes called *Tarskian*. The application of this theorem to semantics was pioneered by Scott [12] and Kahn [8].

Note that the constructive technique given by (6) might be a reasonable implementation of Kahn process networks. Begin with all sequences empty, and start iteratively applying functions. If we choose this constructive technique as the operational semantics, then this theorem tells us that this operational semantics is consistent with the denotational semantics (the least fixed point semantics), so we have full abstraction. For a complete treatment of full abstraction, see Winskel [14].

Under this least-fixed-point semantics, the value of s in figure 2 is λ, the empty signal, when $F = I$. Under this semantics, this is the only sequence that satisfies the composite process, so the composite process is determinate. Intuitively, this solution agrees with a reasonable execution of the process, in which we would not produce any output from $F = I$ because there are no inputs.

Another fixed-point theorem deals with monotonic processes that are not necessarily continuous. This theorem states that a monotonic function on a CPO has a unique least fixed point, but gives no constructive way to find the least fixed point (see [6], page 96). Fortunately, this lack of constructive solution is not a problem in practice since practical monotonic processes are invariably continuous, at least in the context of Kahn process networks.

2.5 PRACTICAL OPERATIONAL SEMANTICS — SCHEDULING

There are serious practical problems with choosing (6) as the operational semantics. First, the functions that need to be iteratively applied map entire sequences into entire sequences. If any of these sequences becomes infinite, the computation of a single function will not terminate, precluding iterative application. This happens immediately if one of the functions happens to be a source (e.g. $F: S^0 \to S^1$) with an infinite output. In practice, we need to partially compute the functions, carefully controlling the length of the sequences. Sometimes it is possible to store only finite windows into potentially infinite sequences and execute a process network in bounded memory. For a complete and up-to-date exposition on these scheduling issues, see Parks [11].

2.6 HIGHER-ORDER FUNCTIONS

Using the fact that compositions like those in figure 1 preserve continuity, and that feedback as in figure 2 is determinate under a least-fixed-point semantics, we can conclude that arbitrary finite compositions of continuous functions are determinate under such semantics. This determinacy, however, is not constrained to bounded or static compositions, where the number of sequences and functions is *a-priori* determined. To generalize the result, we can view the compositions of figures 1 and 2 to be examples of higher order functions. Higher-order functions are functions that take functions as arguments and return functions. We can define a CPO over functions, and arrive at a very powerful generalization of the determinacy result. We will then use a similar technique to study dataflow with firing (Dennis dataflow).

Consider the set of all functions $F : S^m \to S^n$. The prefix order on sequences induces an ordering on functions in this set. We write $F \sqsubseteq F'$ if for all $\mathbf{s} \in S^m$, $F(\mathbf{s}) \sqsubseteq F'(\mathbf{s})$. This is just a pointwise extension of the prefix ordering. Denote the set of functions with the pointwise prefix order by $(S^m \to S^n)$. It is a CPO. To show this, we need to show that all chains in $(S^m \to S^n)$ have a LUB in $(S^m \to S^n)$. Consider such a chain,

$$F_0 : S^m \to S^n,\ F_1 : S^m \to S^n,\ \ldots \tag{7}$$

where $i \le j \Rightarrow F_i \sqsubseteq F_j$. Let $\mathbf{s}$ be any tuple of sequences in S^m. Note that $i \le j \Rightarrow F_i(\mathbf{s}) \sqsubseteq F_j(\mathbf{s})$, so $F_0(\mathbf{s}), F_1(\mathbf{s}), \ldots$ is a chain in S^n. Since S^n is known to be a CPO, this chain has a LUB. Define the function $F : S^m \to S^n$ by

$$F(\mathbf{s}) = \sqcup \{F_0(\mathbf{s}), F_1(\mathbf{s}), \ldots\}. \tag{8}$$

In the pointwise prefix order, this can be written

$$F = \sqcup \{F_0, F_1, \ldots\}. \tag{9}$$

Thus, every such chain has a LUB, so the set of functions with the pointwise prefix order is a CPO.

Following Davey and Priestley [6], let $[S^m \to S^n]$ denote the set of *continuous* functions mapping S^m into S^n ordered by the same pointwise prefix order. Clearly, $[S^m \to S^n] \subset (S^m \to S^n)$. Moreover, $[S^m \to S^n]$ is itself a CPO ([6], theorem 3.17), so it is called a sub-CPO of $(S^m \to S^n)$. This means that any chain of continuous functions has a least upper bound that is also a continuous function.

The bottom element of both $(S^m \to S^n)$ and $[S^m \to S^n]$ is a function $\psi : S^m \to S^n$ that always returns Λ, an n-tuple of empty sequences.

We can of course define a set $(S^m \to S^n)^{\alpha}$ of α-tuples of functions, where $\alpha \in N$. This is also a CPO under an elementwise extension of the pointwise prefix order. The bottom element of this CPO, which we denote by Ψ, is an α-tuple of functions ψ. Finally, we can define sets mixed tuples, for example $(S^m \to S^n) \times (S^p \to S^q)$. Any member of this set is a 2-tuple where the first element is a function from $(S^m \to S^n)$ and the second element is a function from $(S^p \to S^q)$.

Consider a mapping $\phi : (S^m \to S^n) \to (S^p \to S^q)$. Such mappings are sometimes called *functionals* because they map functions to functions. A functional ϕ is *monotonic* if $F \sqsubseteq F'$ implies that $\phi(F) \sqsubseteq \phi(F')$. It is *continuous* if for every chain $W \subset (S^m \to S^n)$, $\phi(W)$ has a least upper bound $\sqcup \phi(W)$, and

$$\phi(\sqcup W) = \sqcup \phi(W). \tag{10}$$

The notation $\phi(W)$ denotes a set obtained by applying the mapping ϕ to each element of W.

The compositions of figure 1 can be described as functionals over this new CPO of functions. Beginning with figure 1(a), denote the composition by $F = \phi(F_1, F_2)$. The relevant mappings are

$F_1 : S \to S$, $F_2 : S \to S$, $F : S^2 \to S^2$, and $\phi : (S \to S)^2 \to (S^2 \to S^2)$. An alternative notation that might be more familiar to some readers would write $F = F_1 \times F_2$, the tensor products in Abramsky's interaction categories [1]. It is straightforward to show that ϕ is continuous over the CPO of functions.

Figure 1(b) is little different from figure 1(a). The only difference is that F_2 is replaced by an identity function. In this case, we can write $F' = \phi'(F_1)$, where $F_1 : S \to S$, $F' : S^2 \to S^2$, and $\phi' : (S \to S) \to (S^2 \to S^2)$. Alternatively, we can write this using the tensor product notation, $F' = F_1 \times I$, where $I : S \to S$ is the identity function.

Figure 1(c) represents a composition of functions. This can also be described as a functional, $F'' = \phi''(F_1, F_2) = F_2 \bullet F_1$. The relevant mappings here are $F_1 : S \to S$, $F_2 : S \to S$, $F'' : S \to S$, and $\phi'' : (S \to S)^2 \to (S \to S)$. With this choice, the sequence s_2 between the two functions is hidden by the composition (it is not exposed as an output). It is also straightforward to show that this functional is continuous.

Using functionals like ϕ, ϕ', and ϕ'' above, appropriately generalized to operate on functions with various numbers of input and output sequences, a rich set of compositions can be constructed. When doing this, it is usually more convenient to use the tensor product notation, $F = F_1 \times F_2$, and the function composition notation, $F'' = F_2 \bullet F_1$. For example, the composition in figure 1(d) can be given by the functional $\phi''' : ((S^2 \to S) \times (S \to S^2)) \to (S^2 \to S^2)$, where $\phi''' = (I \times F_2) \bullet (F_1 \times I)$.

More interestingly, such functionals make it possible to describe unbounded and data-dependent compositions of processes. A classic example, used by Kahn and MacQueen [9], is the sieve of Eratosthenese, which given the input sequence $[2, 3, 4, \ldots]$ outputs the prime numbers. Assume

$V = \{2, 3, 4, \ldots\}$ is the set of possible token values, and let $G_v(s)$ be a "filter" process that, given any sequence s of tokens in V, outputs a subsequence consisting only of those tokens in s that are not multiples of v, for some $v \in V$. The sieve of Eratosthenese is constructed by composing such filters, one for each prime number. Formally,

$$F(s) = \begin{cases} \lambda \quad \text{if} \quad s = \lambda \\ [v].F(G_v(s')) \quad \text{if} \quad s = [v].s' \quad \text{for some } v \in V, s' \in S \end{cases}. \tag{11}$$

This recursive definition is easy to understand if we assume that the input is the sequence $s = [2, 3, 4, \ldots]$ and examine the first few unravelings of the recursion:

$$\begin{aligned} &2.F(G_2([3, 4, \ldots])) \\ &2.3.F(G_3(G_2([4, 5, \ldots]))) \\ &2.3.5.F(G_5(G_3(G_2([6, 7, \ldots])))) \end{aligned} \tag{12}$$

Notice that in effect, the recursion specified a cascaded composition of filters, one for each prime that has been discovered so far. The composition grows dynamically as more primes are discovered.

3. Dataflow with Firing

Continuous functionals on posets of functions provide a convenient way to study Dennis dataflow, which is equivalent to Kahn process networks where processes are made up of a sequence of atomic computations called *firings*. The firings themselves can be described as functions, and the invocation of these firings is controlled by *firing rules*. We can now make this precise.

3.1 DATAFLOW ACTORS

First, we need a little more notation. A tuple $\mathbf{s} \in S^m$ of sequences is said to be *finite* if each of the sequences in the tuple has a finite number of tokens. If $\mathbf{s} \in S^m$ is finite and $\mathbf{s}' \in S^m$ is some other tuple of sequences, then $\mathbf{s}.\mathbf{s}'$ is the *concatenation* of the two tuples of sequences. This is constructed in the obvious way so that $\mathbf{s} \sqsubseteq \mathbf{s}.\mathbf{s}'$. Concatenation was defined above for single sequences rather than tuples.

Given two tuples of sequences $\mathbf{s}, \mathbf{s}' \in S^m$, their *join*, written $\mathbf{s} \sqcup \mathbf{s}'$, if it exists, is defined to be least upper bound of the two sequences. If the join exists, $\mathbf{s}$ and $\mathbf{s}'$ are said to be *joinable*.

We begin with a simpler definition that excludes some useful cases, and then generalize. A *dataflow actor* with m inputs and n outputs is a pair $\{f, R\}$, where:

1. $f: S^m \to S^n$ is a function mapping called the *firing function*,
2. $R \subset S^m$ is a set of finite sequences called the *firing rules,*
3. $f(\mathbf{r})$ is finite for all $\mathbf{r} \in R$, and
4. no two distinct $\mathbf{r}, \mathbf{r}' \in R$ are joinable.

The last constraint implies that for any given $\mathbf{s} \in S^m$ there is at most one $\mathbf{r} \in R$ such that $\mathbf{r} \sqsubseteq \mathbf{s}$. If there is such an $\mathbf{r}$, then there will be a unique $\mathbf{s}' \in S^m$ such that $\mathbf{s} = \mathbf{r}.\mathbf{s}'$.

3.2 DATAFLOW PROCESSES

A Kahn process F based on the dataflow actor $\{f, R\}$ can now be defined as follows:

$$F(\mathbf{s}) = \begin{cases} f(\mathbf{r}).F(\mathbf{s}') & \text{if there exists } \mathbf{r} \in R \text{ such that } \mathbf{s} = \mathbf{r}.\mathbf{s}' \\ \Lambda & \text{otherwise} \end{cases} . \tag{13}$$

Notice that this definition is self-referential. It is by no means obvious that a function F exists that satisfies (13), nor is it obvious that this function is unique. We will show that such a function exists, and that there is a unique least function in the pointwise prefix order. Consider the functional

$\phi : (S^m \to S^n) \to (S^m \to S^n)$ associated with a particular dataflow actor $\{f, R\}$ defined as follows,

$$(\phi(F))(\mathbf{s}) = \begin{cases} f(\mathbf{r}).F(\mathbf{s}') & \text{if there exists } \mathbf{r} \in R \text{ such that } \mathbf{s} = \mathbf{r}.\mathbf{s}' \\ \Lambda & \text{otherwise} \end{cases} . \tag{14}$$

Theorem 1: The functional ϕ is monotonic.

Proof: Consider a particular $\mathbf{s} \in S^m$. We consider two cases. First, assume that there exists one $\mathbf{r} \in R$ such that $\mathbf{r} \sqsubseteq \mathbf{s}$ and $\mathbf{s} = \mathbf{r}.\mathbf{s}'$. In this case, $(\phi(F))(\mathbf{s}) = f(\mathbf{r}).F(\mathbf{s}')$ and $(\phi(F'))(\mathbf{s}) = f(\mathbf{r}).F'(\mathbf{s}')$ for any two functions $F, F' : S^m \to S^n$. If $F \sqsubseteq F'$, then clearly

$$(\phi(F))(\mathbf{s}) \sqsubseteq (\phi(F'))(\mathbf{s}) . \tag{15}$$

Second, suppose that there is no $\mathbf{r} \in R$ such that $\mathbf{r} \sqsubseteq \mathbf{s}$. Then $(\phi(F))(\mathbf{s}) = (\phi(F'))(\mathbf{s}) = \Lambda$ for any two functions $F, F' : S^m \to S^n$. Again, if $F \sqsubseteq F'$, then (15) holds. Thus, (15) holds for any $\mathbf{s} \in S^m$ and F, F' such that $F \sqsubseteq F'$, implying that ϕ is monotonic.

Since the functional ϕ given in (14) is a monotonic function over a CPO, it has a least fixed point F such that $\phi(F) = F$ [6]. This least fixed point satisfies (13), so we take it to be the semantics of the dataflow process.

The existence of a least fixed point is reassuring, but we can go a step further and give a constructive procedure for finding that least fixed point. Moreover, this constructive procedure will exactly match a reasonable operational semantics for single dataflow actors.

Theorem 2: The functional ϕ given by (14) is continuous.

Proof: Consider any chain $\mathbf{F} \subset (S^m \to S^n)$. We need to show that $\phi(\sqcup \mathbf{F}) = \sqcup \phi(\mathbf{F})$. Write $\mathbf{F} = \{F_0, F_1, \ldots\}$ and note that for any $\mathbf{s} \in S^m$

$$\sqcup(\phi(\mathbf{F}))(\mathbf{s}) = \sqcup\{(\phi(F_0))(\mathbf{s}), (\phi(F_1))(\mathbf{s}), \ldots\}. \tag{16}$$

Since ϕ is monotonic, this is a chain in S^n, a CPO, and therefore has a LUB. There are two cases to consider. First, if there exists an $\mathbf{r} \in R$ such that $\mathbf{r} \sqsubseteq \mathbf{s}$ and $\mathbf{s} = \mathbf{r}.\mathbf{s}'$, then

$$\sqcup(\phi(\mathbf{F}))(\mathbf{s}) = \sqcup\{f(\mathbf{r}).F_0(\mathbf{s}'), f(\mathbf{r}).F_1(\mathbf{s}'), \ldots\} = f(\mathbf{r}).\sqcup\{F_0(\mathbf{s}'), F_1(\mathbf{s}'), \ldots\}$$

$$= f(\mathbf{r}).(\sqcup\mathbf{F})(\mathbf{s}') = (\phi(\sqcup\mathbf{F}))(\mathbf{s}). \tag{17}$$

The second case we need to consider is where there is no $\mathbf{r} \in R$ such that $\mathbf{r} \sqsubseteq \mathbf{s}$. In this case,

$$\sqcup(\phi(\mathbf{F}))(\mathbf{s}) = \sqcup\{\Lambda, \Lambda, \ldots\} = \Lambda = (\phi(\sqcup\mathbf{F}))(\mathbf{s}). \tag{18}$$

Thus, in both cases, $\phi(\sqcup\mathbf{F}) = \sqcup\phi(\mathbf{F})$, so ϕ is continuous.

Since ϕ is continuous, not only does it have a least fixed point, but there is a constructive procedure for finding that least fixed point [6]. We start with the "bottom" of the poset, which in this case is the bottom function $\psi : S^m \to S^n$ that always returns Λ, an n-tuple of empty sequences. Let $F_0 = \psi$, $F_1 = \phi(F_0)$, $F_2 = \phi(F_1)$, etc. This forms a chain, and the LUB of this chain is the least fixed point of ϕ.

Examining this chain more closely, suppose for a given $\mathbf{s} \in S^m$ there is a sequence $\mathbf{r}_1, \mathbf{r}_2, \ldots \in R$, such that $\mathbf{s} = \mathbf{r}_1.\mathbf{r}_2.\ldots$. Then the chain takes the following form:

$$\begin{aligned} &F_0(\mathbf{s}) = \Lambda \\ &F_1(\mathbf{s}) = f(\mathbf{r}_1) \\ &F_2(\mathbf{s}) = f(\mathbf{r}_1).f(\mathbf{r}_2) \\ &\ldots \end{aligned} \tag{19}$$

This exactly describes the operational semantics of Dennis dataflow for a single actor. It says to start with each actor producing the empty sequence. Then find the prefix of the input that matches a firing rule, and invoke the firing function on that prefix, producing a partial output. Because of condition (4) on the firing rules, no more than one firing rule can match. Then find the prefix of the remaining inputs that match another firing rule, invoke the firing function on that prefix, and concatenate the result with the output.

In general, it is possible that even if $\mathbf{s}$ is infinite, there will only be a finite sequence $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_p \in R$, for some natural number p, such that $\mathbf{s} = \mathbf{r}_0.\mathbf{r}_1.\ldots.\mathbf{r}_P.\mathbf{s}'$, and $\mathbf{s}'$ will have no prefix in R. In both our operational and our denotational semantics, the firings simply stop, and the output is finite.

3.3 CONTINUITY OF THE DATAFLOW PROCESS

The function F defined by (13) is the least fixed point of the continuous functional defined by (14). For a given input $\mathbf{s}$, the value of $F(\mathbf{s})$ is the least upper bound of the chain given by (19). This chain will be finite for some $\mathbf{s}$ (certainly for finite $\mathbf{s}$, but also for any $\mathbf{s}$ for which after some point, no more firing rules match), and infinite for other $\mathbf{s}$. Since each $F_i \in [S^m \to S^n]$ is a continuous function, and the set $[S^m \to S^n]$ of continuous functions is a CPO, then the LUB F is continuous, and hence describes a valid Kahn process that guarantees determinacy. Note that the firing function f need not be continuous. In fact, it does not even need to be monotonic. It merely needs to be a function defined and finite for each of the firing rules.

3.4 EXAMPLES OF FIRING RULES

Consider for example a system where the set of tokens is $V = \{0, 1\}$. Let us examine some possible firing rules $R \subset S^2$ for unary firing functions $f: S \to S$. We will denote a sequence of tokens using square brackets and commas, so [0,1,1] is a sequence with three tokens. An empty sequence will be denoted with λ, as usual. A set of tuples will be denoted using the usual braces for sets. The following sets of firing rules all satisfy condition (4) above, that no two distinct $\mathbf{r}, \mathbf{r}' \in R$ are joinable:

$$\begin{array}{c} \{\lambda\} \\ \{[0]\} \\ \{[0], [1]\} \\ \{[0, 0], [0, 1], [1, 0], [1, 1]\} \end{array} . \tag{20}$$

The first of these corresponds to a function that consumes no tokens from its input sequence, and can fire infinitely regardless of the length of the input sequence. The second consumes only the leading zeros from the input sequence, and then stops firing. The third consumes one token from the input on every firing, regardless of its value. The fourth consumes two tokens on the input on every firing, again regardless of the values. An example of a set of firing rules that does not satisfy condition (4) is:

$$\{\lambda, [0], [1]\} . \tag{21}$$

Such firing rules would correspond to an actor that could nondeterministically consume or not consume an input token upon firing.

The firing rule in (21) would also be the firing rule of the unit delay defined in Section 2.2, so such a unit delay cannot be a dataflow actor with such firing rules. In fact, delays in dataflow process networks are usually implemented directly as initial tokens on an arc, rather than trying to use sequences of firings, so if we admit such implementation, then there is no loss of generality here. The run-time cost is lower, and this strategy avoids having to have special firing rules for delays that, if allowed in general, could introduce nondeterminism. Note further that once we admit the unit delay, it is easy to model actors with state using a single self-loop initialized to the initial state.

Let us examine some possible sets of firing rules $R \subset S^2$ for binary firing functions $f: S^2 \to S$. A tuple of tokens will be denoted using parentheses, as in $([1], [0])$, a 2-tuple with two sequences of length 1. The following firing rules all satisfy condition (4):

$$\begin{gathered}\{([0],[0]),([0],[1]),([1],[0]),([1],[1])\} \\ \{([0],\lambda),([1],[0]),([1],[1])\} \\ \{([0],\lambda),([1],\lambda)\} \\ \{([1],[0],\lambda),([0],\lambda,[1]),(\lambda,[1],[0])\}\end{gathered} \tag{22}$$

The first of these corresponds to an actor that consumes one input token from each of two inputs. This could implement, for example, a logic function such as AND or OR. The second corresponds to a conditional actor where the first input provides a control token on every firing, and if the control token is "1", then a token is consumed from the second input. Otherwise, no token is consumed from the second input. The third corresponds to an actor that never consumes a token from the second input. The last corresponds to the famous Gustave function [4]. It is a particularly interesting set of firing rules because it cannot be implemented with the blocking reads of the Kahn-MacQueen implementation of Kahn process networks [9].

The following firing rules do not satisfy condition (4):

$$\{([0],\lambda),([1],\lambda),(\lambda,[0]),(\lambda,[1])\} \ . \tag{23}$$

Such would be the firing rules of the famous *nondeterminate merge*, a process that can consume a token on either input and copy it to its output. The nondeterminate merge is not a monotonic process, and so use of it in a Kahn process network could result in nondeterminism.

While actors satisfying conditions (1) through (4) above result in continuous Kahn processes, these conditions are more restrictive than what is really necessary. The firing rules in (23), for example, are not only the firing rules for the dangerous nondeterminate merge, but are also the firing rules for a perfectly harmless two-input, two-output identity function, $I(\mathbf{s}) = \mathbf{s}$ for all $\mathbf{s} \in S^2$. It might seem at first glance that such an identity function could be implemented using the first firing rule of (22), but in fact this will not work. The two examples in figure 3 show why not. In the first of these examples, the first (top) input and output should be the empty sequence, λ, under the least-fixed-point semantics, so there will never be a token to trigger the firing rule of (22). In the second of these examples, the second (bottom) input and output have the same problem. The firing rules of (23), however, have no difficulty with these cases. In the next subsection we replace rule (4) with a more general rule.

3.5 COMMUTATIVE FIRINGS AND COMPOSITIONALITY

Many dataflow models with a notion of firing are not compositional. That is, an aggregation of actors that can be individually described using firings cannot be collectively described using firings. This problem was alluded to in the final example of the last section, which is the simplest example illustrating the problem. The two-input, two-output identity function can be thought of as a aggregation of two one-input, one-output identity functions, as suggested in figure 4. One-input, one-output identity functions are trivially described as dataflow actors satisfying constraints (1) through (4), but the two-input, two-output identity cannot be so described.

To solve this problem, we can replace rule (4) with the following more complicated rules:

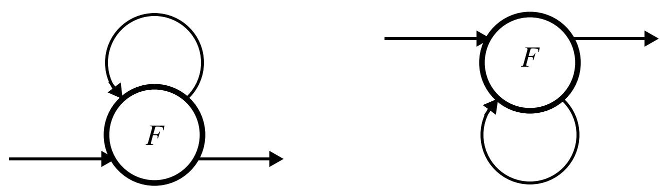

FIGURE 3. If *F* is an identity function, the appropriate firing rules are given in (23).

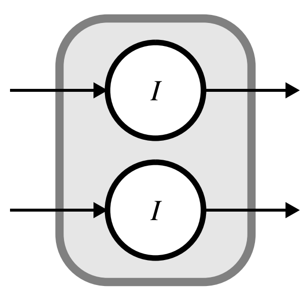

FIGURE 4. A two-input, two-output identity function described as an aggregation of two one-input, one-output identity functions.

5. For any $\mathbf{r}, \mathbf{r}' \in R$ that are joinable, $\mathbf{r} \sqcap \mathbf{r}' = \Lambda$ (the greatest lower bound is the tuple of empty signals) and $f(\mathbf{r}).f(\mathbf{r}') = f(\mathbf{r}').f(\mathbf{r})$.
6. If $\Lambda \in R$ then $f(\Lambda) = \Lambda$.

This rule generalizes the previous model by allowing a set of firing rules where more than one firing rule can match the inputs. However, if more than one firing rule matches the inputs, then it should make no difference in what order these firing rules are used. The firing function, therefore, applied to these joinable firing rules, should commute with respect to the concatenation operator.

We will also need to redefine the functional that we used to construct the Kahn process from the dataflow actor. To do this conveniently, let $P_R(\mathbf{s})$ denote the set of all firing rules $\{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_p\} \subset R$ that are prefixes of $\mathbf{s}$. This set could be empty if there are no prefixes of $\mathbf{s}$ in R. Then define the functional

$$(\phi'(F))(\mathbf{s}) = \begin{cases} f(\mathbf{r}_1).f(\mathbf{r}_2). \ldots .f(\mathbf{r}_p).F(\mathbf{s}') \text{ for } \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_p\} \in P_R(\mathbf{s}) \text{ if } P_R(\mathbf{s}) \neq \varnothing \\ \Lambda \text{ otherwise} \end{cases} . \quad (24)$$

Note that because of property (5), it makes no difference in what order we use the matching the firing rules $\{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_p\}$. In the above, $\mathbf{s}'$ is defined by $\mathbf{s} = \mathbf{r}_1.\mathbf{r}_2.\ldots.\mathbf{r}_p.\mathbf{s}'$.

Although the notation gets a bit more tedious, it is straightforward to extend the above results to conclude that the functional ϕ' and the function F that is its least fixed point are continuous. The proofs are very similar to that above.

3.6 PRACTICAL OPERATIONAL SEMANTICS — SCHEDULING

The constructive procedure given by (19) ensures that repeated firings converge to the appropriate Kahn process defined by the actor. If any such sequence of firings is finite, then it is only necessary to invoke a finite number of firings. In practice, it is common for such firing sequences to be infinite, in which case a practical issue of fairness arises. In particular, since there are usually many actors, in order to have the operational semantics match the denotational semantics, it is necessary that the firing function of each actor occur infinitely often, if possible.

It turns out, however, that such a fairness condition is not always desirable. It may result in unbounded memory requirements for execution of a dataflow process network. In some such cases, there is an alternative firing schedule that is infinite but requires only bounded memory. That firing schedule may not match the denotational semantics, and may nonetheless be preferable to one that does.

A simple example is shown in figure 5. The actor labeled "SELECT" has the firing rule (again assuming $V = \{0, 1\}$)

$$\{([1], \lambda, [1]), ([0], \lambda, [1]), (\lambda, [1], [0]), (\lambda, [0], [0])\}, \tag{25}$$

where the order of the inputs is top-to-bottom. If the bottom input (the control input) has value "1" (for TRUE), then a token of any value is consumed from the top input, and no token is consumed from the middle input. If the control input has value "0" (for FALSE), then no token is consumed from the top input, and a token of any value is consumed from the middle input.

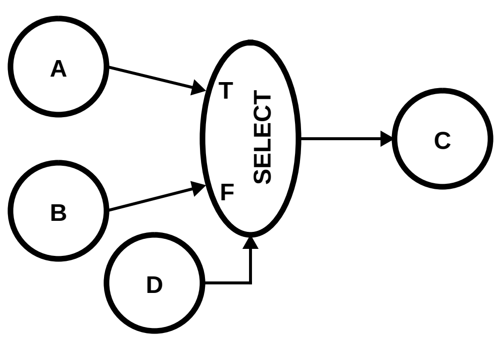

FIGURE 5. An example of a dataflow process network where it may be undesirable from a practical perspective to insist that the operational semantics match the denotational semantics.

Suppose that the actors A, B, and D, all of which are sources (i.e. of type $F : S^0 \to S^1$), are defined to each produce an infinite sequence, and that C (which is of type $F : S^1 \to S^0$), is defined to consume an infinite sequence with any token values. Suppose further that the output from D is the constant sequence $[0, 0, 0, \ldots]$. Then tokens produced by actor A will never be consumed. In most practical scenarios, it is preferable to avoid producing them if they will never be consumed, despite the fact that this violates the denotational semantics, which state that the output of actor A is an infinite sequence. This problem is solved by Parks [11], who shows that the obvious solution for the example in figure 5, demand-driven execution, does not solve the problem in general.

4. Acknowledgments

I wish to acknowledge useful discussions with Gerard Berry, Shuvra Bhattacharyya, Praveen Murthy, Dick Stevens, and the students of EE290n in the Fall of 1996 at Berkeley, who greatly helped me to refine this material.

This work was partially supported under the Ptolemy project, which is sponsored by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Dolby Laboratories, Hitachi Ltd. and Hitachi America, LG Electronics, Mitsubishi, Motorola, NEC, Philips, and Rockwell.

5. References

[1] S. Abramsky, S. J. Gay, and R. Nagarajan, "Interaction Categories and the Foundations of Typed Concurrent Programming," In: *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School,* (M. Broy, ed.), NATO ASI Series F, Springer-Verlag, 1995.

[2] Arvind, L. Bic, T. Ungerer, "Evolution of Data-Flow Computers," in *Advanced Topics in Data-Flow Computing,* ed. J.-L. Gaudiot and L. Bic, Prentice-Hall, 1991.

[3] A. Benveniste, P. Caspi, P. Le Guernic, N. Halbwachs, “Data-flow Synchronous Languages,” in J. W. de Bakker W.-P. de Roever, and G. Rozenberg, eds., *A Decade of Concurrency — Reflections and Perspectives*, Lecture Notes in Computer Science no. 803, Springer-Verlag, Berlin, 1994.

[4] G. Berry, “Bottom-Up Computation of Recursive Programs,” *Revue Française d’Automatique, Informatique et Recherche Opérationnelle*, vol. 10, no. 3, pp. 47-82, March, 1976.

[5] G. Birkhoff and S. MacLane, A survey of modern algebra, 4th ed., Macmillan, New York, 1977.

[6] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[7] J. B. Dennis, “First Version Data Flow Procedure Language”, Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.

[8] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[9] G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[10] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, May 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)

[11] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. **PhD Dissertation**. EECS Department, University of California. Berkeley, CA 94720, December 1995. (http://ptolemy.eecs.berkeley.edu/papers/parksThesis)

[12] D. Scott, “Outline of a mathematical theory of computation”, *Proc. of the 4th annual Princeton conf. on Information sciences and systems*, 1970, 169-176.

[13] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* The MIT Press, Cambridge, MA, 1977.

[14] G. Winskel, *The Formal Semantics of Programming Languages*, the MIT Press, Cambridge, MA, USA, 1993.