McCarthy's Amb Cannot Implement Fair Merge

Prakash Panangaden
Vasant Shanbhogue

88-913
May 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

McCarthy's Amb Cannot Implement Fair Merge

Prakash Panangaden
Vasant Shanbhogue
Computer Science Department, Cornell University

May 3, 1988

Abstract

In this paper we establish that **fair merge** is a powerful non-deterministic primitive that cannot be implemented in terms of other well-known primitives. It is well known that fair merge embodies countable non-determinism. It is also been known that McCarthy's **amb** embodies countable non-determinism. It had not been known, however, whether **amb** could implement a **fair merge**. We show that countable non-determinism together with angelic non-determinism cannot implement fair merge even in dynamic dataflow networks. This settles a question posed by Abramsky over four years ago. Earlier work had suggested this result by showing that for *static* dataflow networks, one cannot implement **fair merge** using angelic merge.

1 Introduction

The theory of computability is well established for sequential, determinate programs. When one considers non-determinism in the context of a parallel programming language, the situation is much less clear. We examine the relative expressive power of non-determinate primitives in the context of dynamic dataflow networks. Our main result is that it is impossible to implement a **fair merge** using so called "angelic non-determinism." The key point of the result is that one cannot implement **fair merge** without some sort of time sensitivity built into the primitives.

The results described in the present paper were suggested by earlier results of Panangaden and Stark [17] that applied to static networks, i.e. to networks whose structure is fixed throughout execution. The techniques used in [17] were operational in nature. Similar results were proved in [16] using denotational techniques and continuity arguments. We use a different denotational model suggested by [5].

A semantic theory appropriate for finite non-determinism was developed by Plotkin [20] using powerdomains. Plotkin's powerdomain techniques cannot handle countable non-determinism. This motivated people to explore techniques that were capable of handling countable non-determinism. Work along these lines was done by Back [4], Park [19,18], Abramsky [1], Keller and Panangaden [9,10] and Panangaden [15].

A powerdomain technique for countable non-determinism has been studied by Apt and Plotkin [3]. They develop a powerdomain for countable non-determinism when the underlying domain is flat. Their semantics is for a sequential, imperative language. They observe that one can program fairness using countable non-determinism. In the case of parallel dataflow programs, however, the situation is complicated by the fact that a fair merge not only has to choose fairly, but has to avoid waiting forever for data on an empty channel. In this situation, fairness is not the same as countable non-determinism as is shown in [17]. In the present paper we show that countable nondeterminism, the ability to detect and avoid non-terminating computation, and the ability to define dynamically expanding networks, *even combined*, do not suffice to implement a fair merge.

1.1 The Basic Model

We quickly recall the basic model of static dataflow computing due to Gilles Kahn [7]. A *static dataflow network* is a directed graph. The vertices represent autonomous computing agents, each one executing a sequential, determinate program. These are called determinate Kahn processes. The arcs represent communication channels between the computing agents. We assume that the channels are unidirectional, unbounded queues. The computing agents communicate with each other, and, in general, with the environment, by passing data values, called "tokens," along the channels. The computing agents interact with the channels by **read** and **write** commands. Unlike CSP or CCS, communication is entirely asynchronous. A **write** is executed unconditionally and a **read** causes a computing agent to suspend until data becomes available.

Under these circumstances, one can define the semantics of a network in a simple way. One associates with each channel a stream, i.e. a potentially infinite sequence of tokens, and with each computing agent, a function from streams to streams. The collection of streams forms a Scott domain under the prefix ordering. One can show that the functions associated with the computing agents are all continuous. The relationship between the streams associated with different channels are expressed equationally. The network may contain feedback loops and, thus, the equations between the streams may be mutually recursive. The least solution is obtained using standard fixed-point theory. The fixed-point solution is the meaning of the network. It can be shown that this agrees with an operational semantics for these so called "Kahn" networks; see for example Stark [24].

The presence of non-determinism complicates the above picture. The behaviour of a computing agent can no longer be expressed as a function. Several semantic theories have been proposed. The point is that one cannot naively use sets of possible streams to represent the meanings of non-determinate networks. This is because causal anomalies arise, as observed by Keller [8] and by Brock and Ackerman [5]. The solution is to use the computational events in the network as the building blocks of the semantic theory. Such an approach was proposed by Keller [8] and Brock and Ackerman [5] and Pratt [23] and also developed by Keller and Panangaden [9,10]. Recently, very interesting work has been done by Joost Kok [11,12] in developing a fully abstract semantics for such nets. For our purposes, we do not need a very elaborate semantic theory; our goal is not to produce the "best" semantics but rather to use one that allows us to make the necessary distinctions. Thus, we need not be concerned whether our semantics is the most refined available. Indeed, it strengthens our results if we can prove inequivalences with a crude semantics.

1.2 Merge Primitives

In this subsection, we summarize the results reported in Panangaden and Stark [17].

By countable nondeterminism or unbounded nondeterminism, we mean the dataflow analogue of the "random" assignment statement $x :=?$. We prefer not to use the word "random" because of the associations with probabilistic computing. The countable nondeterminism process has one input channel and one output channel, and for each value sent out onto the input channel, the value gets read and some positive integer is output.

A **fair merge** primitive has a pair of input channels and a single output channel. Every token that appears at an input channel appears on the output channel eventually. The relative order of tokens from a single input channel are preserved, but tokens from different channels appear interleaved in an unpredictable fashion. An **angelic merge** primitive behaves like **fair merge** except for the fact that if the sequence of tokens on one of the input channels is finite then every token on the other channel is guaranteed to be transmitted to the output channel. Thus, it would behave exactly like **fair merge** if the input sequences were guaranteed to be finite. If the input sequences are infinite then one of the input channels may be ignored in favour of the other one. Intuitively, the **angelic merge** primitive has "the ability to avoid empty channels."

We can consider a primitive that has the "opposite" property, i.e. a merge primitive that is fair if the input streams are guaranteed to be *infinite.* Such a merge is called an "infinity-fair" merge by Park [19]. It turns out that this kind of merge can be implemented by an countably nondeterministic (CN) process by simply using the sequence of integers generated by the CN process as an "oracle" to a process that reads the integers and uses them to determine how many tokens to consume from an input channel before switching to the other input channel. The rather surprising result proved in [17] says that even the combination of angelic merge and infinity-fair merge cannot implement a fair merge.

The method of proof used in [17] to establish this result involves the idea of residuals, from the lambda calculus. Certain maximal computations are identified with respect to an appropriate pre-order and it is shown that completed computations co-incide with the maximal ones. This is then used to show that a particular monotonicity property holds in networks built out of determinate Kahn processes, angelic merge and CN processes. The monotonicity property does not hold for fair merge, thus establishing the non-implementability result. The method used in [16] is based on showing that a monotonicity property is preserved by network composition, using structural induction. This is then used to show that any network composed of determinate Kahn processes, angelic merge and unbounded nondeterminism processes satisfies the monotonicity property. The major problem left unsolved by these approaches was how to apply these arguments to infinte networks such as potentially arise in the dynamic case.

2 The Semantic Model for Static Networks

The semantic model that we use is based on a formalism that encodes the relation of causal precedence between computational events. Such a model was proposed by Keller and was developed by Brock and Ackerman and by Pratt[8,5,22]. Keller and Panangaden [9,10] have developed a trace version of this theory. In an earlier investigation, we used the trace semantics to prove expressiveness results of static dataflow networks [16], but for the dynamic case it turns out to be advantageous to use a model in which event sequences are not interleaved. The formalism we use is essentially that of Brock and Ackerman [5]. In this section, we describe this model for the static case.

For finite static networks, we can prove semantic properties by induction on the structure of networks. So we exploit the compositional nature of the theory. Networks may be composed in three ways. These are aggregation, loop formation and serial composition.

Definition 1 *If M and N are networks, we define a new network, written $M \| N$, called the* **aggregation** *of M and N. The input channels of $M \| N$ are all the input channels of M and N and similarly for the output channels.*

In aggregation, two networks are viewed as a single network, but no new channels are constructed.

In loop formation, a new network is obtained from an old network by connecting an output channel of the old network to an input channel of the old network.

Definition 2 *Given a network N we define the* **loop composition** *of N with b and c to be the network obtained from network N by connecting the output channel b to the input channel c. The notation that we use is $loop(\mathbf{b}, \mathbf{c}, N)$.*

The serial composition of two networks is obtained by connecting the output channel of one with an input channel of the other. It can be viewed as an aggregation followed by a loop formation. We will not consider it further.

A new network may also be formed from an old one by hiding some input and output channels of the old network.

Definition 3 *Given a network N, we define the* **restriction** *of N with respect to the channel a as the network obtained by removing the channel.*

No new channels are constructed during restriction.

During a network computation, the network reads a sequence of data items from each of its input channels, and writes a sequence of data items on each of its output channels.

Definition 4 *A* **history** *of a channel of a network is a sequence of data items transmitted on that channel in some computation of the network.*

We will also refer to the history of a channel c as a c-stream.

As argued by Brock and Ackerman [5], "due to the asynchronous nature of communication, nothing is said about absolute timing or relative timing of events at different operator ports, though the order of values passing through a single port is, of course, significant."

As defined by Brock and Ackerman in [5], a single **scenario** of a network is a set of histories of the input and output channels of the network, one history for each input or output channel, with a partial ordering on the individual data items, or events, of the histories. The set of histories represents a possible, possibly infinite, computation of the network, and the partial ordering shows causality constraints. Two data items are causally ordered under this relation, say a precedes b, if the event of producing a *must* precede the event of producing b. An output item is causally preceded by those input items "which contributed, through a non-indivisible primitive event or through a series of internal process events, to its production."

We first need some definitions.

Definition 5 *A* **vertex** *of a partial order $(A, \rightarrow)$ is any element of A.*

Definition 6 *The* **vertex-set** *of a partial order is the set of its vertices.*

Definition 7 *$a_1, a_2, \ldots a_n$ is said to be a* **path** *from a_1 to a_n in the partial order $(A, \rightarrow)$, if $a_1, \ldots a_n \in A$ and for all i from 1 to $n-1$, $\langle a_i, a_{i+1} \rangle \in (A, \rightarrow)$.*

We model networks in terms of partially ordered sets of event occurrences. We first define an event.

Definition 8 *We define an* **event** *to be a pair consisting of a channel name and a value. An event of the form $(\mathbf{a}, v)$ is interpreted as the passage of the value v on channel $\mathbf{a}$.*

Now we define a scenario formally in terms of partial orders on events.

Definition 9 *A* **scenario** *of a network consists of a partial ordering on the individual events of the history streams on the input and output channels of the network, such that*

(i) events of the history of a single channel are totally related by this ordering, and,

(ii) there is no relationship between events of two different input channels, or of two different output channels, or from output events to input events.

This definition applies to networks with finitely many input and output channels, because that is all that we shall be concerned with. We will be concerned with networks that may have infinitely many internal channels but the number of input and output channels will always be finite.

Lemma 1 *The scenario of any network defines a* well-founded *partial order.*

Proof:
This follows immediately from the definition of scenario and from the fact that histories are well-founded.∎

This lemma is intuitively motivated by the requirement that every event can have only finitely many causes. Winskel [25] observes that this is deeply connected with Scott's thesis that computable functions are continuous.

A scenario may be drawn graphically as follows : for each channel, draw a sequence of data items, corresponding to the history of the channel, and

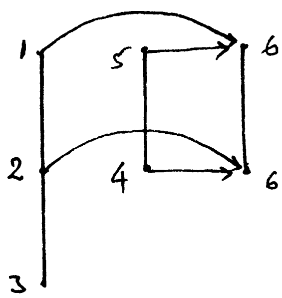

Figure 1: A scenario for **adder**

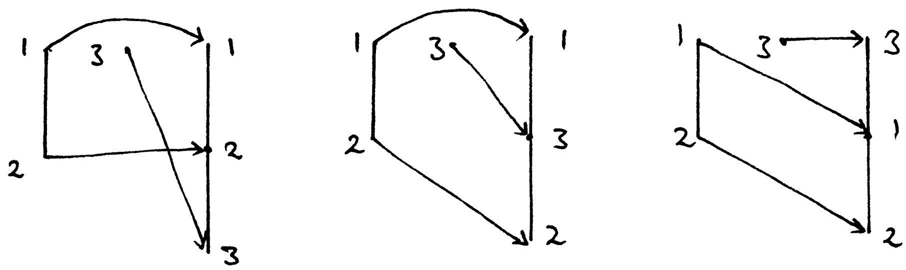

Figure 2: Scenarios for **fair merge**

draw an arrow from each item to the next in this sequence. Also draw arrows from an input item to an output item whenever the input item causally precedes the output item. The requirement that this is a partial order is just the requirement that there is no directed cycle of arrows as this would violate antisymmetry.

Example : Define a process **adder** with two input channels and one output channel. The process reads an integer from each input channel, outputs their sum, and keeps repeating these actions. When the input histories are $(1 \cdot 2 \cdot 3, 5 \cdot 4)$, the scenario representing the computation is shown in Figure 1.

For another example, consider a **fair merge** process with two input channels and one output channel. The process outputs some interleaving of the two input histories. When the input histories are $(1 \cdot 2, 3)$, the possible scenarios are shown in Figure 2.

Definition 10 *The denotation of a network is the set of possible scenarios of that network. We write $\mathcal{S}(N)$ to stand for the set of scenarios of network*

N.

We now describe how composition of networks is expressed in terms of operations on scenario sets.

Definition 11 *Suppose that S_1 and S_2 are scenarios with disjoint sets of events. Their* **union**, *is the scenario obtained by forming the union of the underlying sets of events, and the partial order is the union of the partial orders of S_1 and S_2.*

If N_1 and N_2 are two networks, and a new network N_3 is formed by the aggregation of N_1 and N_2, then $\mathcal{S}(N_3)$ is the set of unions of all pairs (S_1, S_2), where $S_1 \in \mathcal{S}(N_1)$ and $S_2 \in \mathcal{S}(N_2)$.

If N is a network, and a new network N' is formed by loop composition, say the output channel x of network N is connected to the input channel y of network N, then we obtain the scenario set of N' as follows. Discard all scenarios of N in which the y-stream, the history of channel y, is not equal to the x-stream. For each remaining scenario, identify together the y-stream and the x-stream, that is, identify corresponding events in the x-stream and the y-stream. Discard all the scenarios that produce a directed cycle, because these cannot possibly represent a computation of the network N'. Now, for each remaining scenario, remove the data items corresponding to the identified streams (since they do not appear externally for network N'), preserving the partial order. The resulting set characterizes the new network N'.

Example : Define a process **double-adder** with two input channels i_1 and i_2, and two output channels o_1 and o_2. The process first outputs a 0 on o_2, then repeats the following action : read an integer from each input channel and output their sum on both the output channels. A possible scenario is shown in Figure 3. Now connect o_2 and i_2 together to form an internal channel. We show how to get a scenario for the new network in Figure 3.

The remaining operation on networks is restriction. If N is a network, and $o_1, \ldots o_n$ are output channels to be hidden, then the scenarios for the new network are all the scenarios S' such that there exists $S \in \mathcal{S}(N)$, and S' is the scenario with the streams corresponding to $o_1, \ldots o_n$ removed.

If N is a network, and $i_1, \ldots i_m$ are input channels to be hidden, then the scenarios for the new network are all the scenarios S' such that there

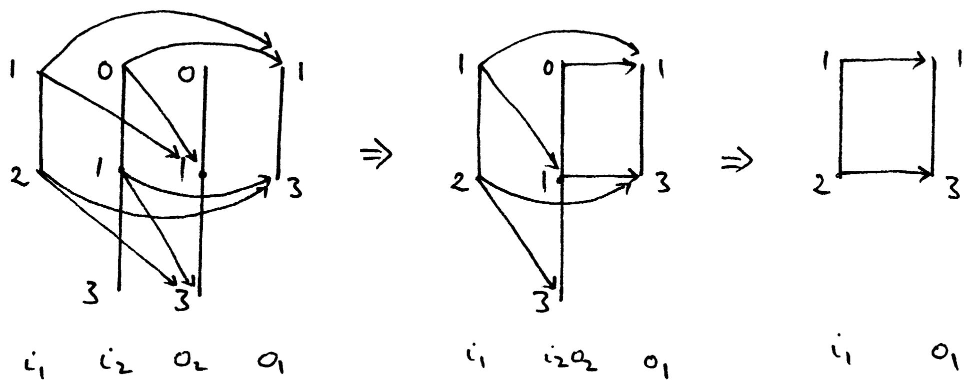

Figure 3: Getting a new scenario

exists $S \in \mathcal{S}(N)$, with the streams corresponding to $i_1, \ldots i_m$ all null, and S' is the scenario with these null streams removed.

The final issue to be discussed in this section is the notion of implementation. We would like a definition that is expressed in terms of *observable* aspects of network behavior. Furthermore we wish to use a very weak notion of implementability. This will make our proofs of non-implementability strong. Of course, if our interest is in implementability results, then we would need a strong notion of implementability.

Definition 12 *Given a network N with C as its set of input and output channels and $\mathcal{S}(N)$ as its scenario set, we define the* **IO-relation** *of N as the set of all C-indexed tuples of histories such that for each tuple there is a scenario in $\mathcal{S}(N)$ that has exactly the same histories.*

This captures the directly observable aspects of a network behavior. The discussion of Brock and Ackerman[5], shows that different networks with the same IO-relation may produce networks with different IO-relations when they are connected to other identical networks. Thus IO-equality is not a congruence with respect to network composition.

Definition 13 *A set S of processes is said to* **implement** *a relation R if there is a finite network, N, built out of copies of the processes in S having R as its IO-relation.*

Even with this very weak notion of implementation, we still cannot implement fair merge with angelic merge. We shall say that a network N

implements network M if they have the same IO-relation, upto renaming of channels. By this definition, we do not care if the two networks have different internal configurations. But they must present the same interface to the external world.

We now define an ordering on scenarios, and use this ordering to define monotonicity and continuity. Monotonicity will turn out to be the key idea in establishing our expressiveness theorem. The ordering captures the idea of "extending" a computation.

Definition 14 *For a given network, if S_1 and S_2 are scenarios, then $S_1 \sqsubseteq S_2$ iff*

(a) for all channels c, the c-stream in S_1 is a prefix of the c-stream in S_2

(b) there does not exist $\langle u, v \rangle \in S_2$, such that $\langle u, v \rangle \notin S_1$ and $v \in$ vertex-set(S_1), and

(c) for any two vertices u and v in vertex-set(S_1), the pair $\langle u, v \rangle \in S_1$ iff $\langle u, v \rangle \in S_2$.

The second condition says that there are no new "causal" paths into old events. The third condition states that all the old causal paths persist in the extended scenario.

Definition 15 *A network is said to be* **monotone** *or* **monotonic** *if, for any scenario S_1 of the network, if the input histories are extended, then there exists a scenario S_2 with the extended input histories, such that $S_1 \sqsubseteq S_2$.*

Definition 16 *A network is said to be* **continuous** *if whenever $S_1 \sqsubseteq S_2 \sqsubseteq \ldots$ is a sequence of scenarios of the network, $\sqcup S_i$ is also a scenario of the network, where vertex-set($\sqcup S_i$) $= \cup_i$vertex-set(S_i), and*

$$\sqcup S_i = \cup S_i.$$

In the above definition, we are viewing a scenario as simply a poset and thus as a set of ordered pairs. When we say union of relations, we mean union of the sets of ordered pairs that are in the relation. Note that the union of a chain of partial orders is automatically a partial order, so the above definition makes sense.

Definition 17 *A network is said to be* **determinate** *if for every tuple of input histories to the network, there is a unique tuple of output histories.*

Processes, as described by Gilles Kahn [7], are determinate in this sense. For our purposes we will take as primitive the notion of determinate Kahn process. We adopt the following definition:

Definition 18 *A* **Kahn process** *is a process with a determinate, monotonic and continuous scenario set.*

We have not formalized the notion of *sequential* process as it is not essential to our proofs. Our results thus apply even if we had non-sequential processes like "parallel or" as described by Plotkin [21].

3 Dynamic Networks

We now come to the main result in this paper. We establish that the results proved by Panangaden and Stark [17] hold when networks contain recursively defined subnetworks that may expand dynamically during network execution. The semantic model is essentially the same as in section 2, except that now we have to include events that correspond to the expansion of a recursive definition.

First of all, we need to clarify what we mean by "recursion." A network X with m input channels and n output channels may be defined as a specific network of processes and also of "black boxes" marked X, with m input channels and n output channels. The idea is that X may expand to the network represented by it, and this network may contain more copies of X. In this way, by suitably defining X, we may generate multiple, even unboundedly many, copies of any processes of interest. X is called a **recursively defined process**. If S is a set of processes, then a **recursively defined** S**-process** X is defined as a network of copies of processes in S and recursively defined S-processes, possibly including X.

Now, we define the notion of *implementation with recursion*, similar to the earlier definition of implementation.

Definition 19 *A set of processes S is said to* **implement** *process B* **with recursion** *if there is a recursively defined S-process with the same IO-relation as B.*

In the presence of recursive definition we can implement both angelic merge and unbounded nondeterminism using a single operator called **amb**. Define an **countably nondeterminate** process (as in section $1 \cdot 2$) to be a process with one input channel and one output channel. The process reads a token from its input channel and outputs an arbitrary positive integer. The process ignores all subsequent input tokens. The **amb** process has two arguments, and selects one or the other. If one of them is a divergent expression then the other one gets selected so **amb** will not wait for a divergent computation if there is a data value available to it. Now consider the function f defined as follows

$$f(n) = \mathbf{amb}(n, f(n+1))$$

If we evaluate this on 0 we get any integer as a result without the possibility of divergence. Thus **amb** implements unbounded nondeterminism. It is easy to write an implementation of angelic merge with **amb**. The unsolved question posed by Abramsky [2] was whether **amb** could implement fair merge. Clearly one can get iteration for free if one has the ability to make recursive definitions. Thus many of the simple arguments that suffice for the static case do not apply here.

Definition 20 *A* **completely expanded network** *is a network in which no recursive definition is left unexpanded.*

Such networks, which represent the maximal "limiting" networks to which networks with recursively defined processes can evolve into, are potentially infinite. Of course, at any finite stage of the execution, the actual dynamic network is finite. We started with a finitely described network, so the resulting completely expanded network has finitely many input and output channels. We now need a way of describing the scenario set of a network in terms of the scenario sets of the countably many components of the network.

We describe the process of constructing the scenarios for a countable network from the scenarios of its components. Let N be a network built up from countably many components $N_1, N_2, \ldots$. Let $S_1, S_2, \ldots$ be scenarios, with $S_i \in \mathcal{S}(N_i)$ for all i, satisfying the following condition: for every internal channel connecting output port y in N_i to input port x in N_j,

N_i not necessarily distinct from N_j, the y-stream in S_i is equal to the x-stream in S_j. Suppose that S' is the preorder obtained by merging together the corresponding streams for internal channels. The preorder S' is not a scenario since it contains events on channels that are internal to the network. We check that S' is well-founded and has no directed cycles. Then S'' is S' with the merged streams removed, i.e. for any u, v not events on any internal stream,

$$\langle u, v\rangle \in S'' \;\Leftrightarrow\; \langle u, v\rangle \in S'$$

The scenario set of the network N consists of all the S'' constructed in this way.

We now prove the main theorem in this paper, i.e. that a network composed of countably many monotonic and continuous components is monotonic. We need to show that if we start with a network scenario and we extend the inputs, then we can get another scenario with appropriately extended outputs. We describe how to construct the scenario with the extended outputs.

Let S be a scenario of the network. Then there exist scenarios S_i for each component such that when we combine them in the manner described above we get the network scenario. Let this be called *Stage 0* in the construction.

Stage 1: Extend the input streams of the network. This causes some of the inputs of the components to get extended. By monotonicity of the components we can extend their scenarios.

Stage 2: Extending the outputs of some of the components causes the input streams to other components to get extended. We use monotonicity to extend the affected component scenarios again.

We continue this process to get a possibly infinite increasing sequence of scenarios for each component. Using continuity of the components, we get a least upper bound (lub) scenario for each component. By construction, all the lub scenarios match up in their channel streams. It now remains to be shown that these lub scenarios for the components can be combined to form a network scenario.

We identify together the appropriate internal streams of the network. If the result of this fails to give a network scenario, then either

(a) there is a directed cycle, or,

(b) there is an infinite descending chain

in the graph representing the resulting poset. We show that neither (a) nor (b) is possible. Actually, absence of (b) means absence of (a) as well, but it is easier to explain if we think of the two possibilities seperately.

We need some preliminary definitions and lemmas.

Definition 21 *At stage n, for every channel connecting output port x to input port y, we know that x-stream $\sqsupseteq$ y-stream by construction. Combine the scenarios of all the components by merging together the y-stream with the prefix of the x-stream equal to the y-stream. Do this for all the internal channels of the network. This preorder is defined to be the* **graph at stage** n.

It represents the partial stage in the construction of the final network scenario.

We will follow the convention that if an event added at stage n is merged with an event added at stage $n+1$, then the resulting event corresponds to stage $n+1$.

Lemma 2 *For every stage, there is no path in the graph at that stage from an event added at that stage to an event added at an earlier stage.*

Proof:
At stage 0, there are no earlier stages, so the statement trivially holds.

Suppose the statement is true upto stage n; consider the graph at stage $n+1$. Consider any event a added at stage $n+1$. If this is an output event, then the only paths from it will be to output events, if there are any, succeeding this event on the same channel. All of these are added at stage $n+1$, not at an earlier stage. If a is an input event, then the only paths from it will be to input events, if there are any, succeeding this event on the same channel, and to new output data events, from which there is no path to an old event. There cannot be any path from a to an old event of the component scenario containing a at stage $n+1$, because of the way scenarios get extended. This proves the desired statement. ∎

Lemma 3 *For every stage n, the graph at stage n has no cycle of directed edges.*

Proof:
At stage 0, there is no cycle, because the component scenarios can be combined to get a network scenario. Suppose the graph is acyclic upto stage n, and there is a directed cycle in the graph at stage $n + 1$.

Case (i) : If the cycle only consists of events added by stage n, then all the edges of the cycle must have been present at stage n. This is because no new edge can get added at stage $n + 1$ into events already present at stage n by the above lemma. This gives a contradiction because then there is a cycle in the graph for stage n.

Case (ii) : If the cycle has some new events (added at stage $n + 1$) and some events added by stage n, then there are events e_1 and e_2 such that e_1 precedes e_2, e_1 added at stage $n + 1$, e_2 added by stage n. This means that a new edge must have got added at stage $n + 1$ into an event already present. By above lemma, this is a contradiction.

Case (iii) : Suppose the cycle has only new events. From a new output event, there are paths only to other output data events succeeding this event on the same channel. Since the data events on a channel are totally ordered, there cannot be a cycle from the output event. From a new input data event, there are paths only to other input data events succeeding this event on the same channel, and to new output data events, from which there cannot be a cycle. Thus this possibility is ruled out as well. ∎

Lemma 4 *For every stage n, the graph at stage n is well-founded, that is, there is no infinite descending chain.*

Proof:
At stage 0, the graph is well-founded, because the component scenarios can be combined to obtain a network scenario. Suppose the graphs at all the stages upto n are well-founded. Consider the graph at stage $n + 1$.

Suppose there is an infinite descending chain of data events $e_0 \leftarrow e_1 \leftarrow e_2 \leftarrow \ldots$. If e_0 was added at an earlier stage, say stage m, then using lemma 1, the entire chain is in the graph at stage m, which gives a contradiction by using the inductive hypothesis.

So e_0 was added at stage $n + 1$ and the entire chain is in the graph at stage $n + 1$. Now all the events on the chain must have been added at stage $n + 1$, because if any e_i was added at an earlier stage, then the infinite desending chain $e_i \leftarrow \ldots$ would have been in the graph at an earlier stage, contradicting the inductive hypothesis. We note that each history is well-founded. If e_0 is an output data event, then the only new data events that can precede it are the finitely many new output data events preceding e_0 on the same channel and new input data events. So, since there are only finitely many new output data events that can precede e_0 , there is an e_i which is a new input data event. But the only new data events that can precede e_i are the finitely many input data events preceding e_i on that channel. This contradicts the assumption that there is an infinite descending chain. ∎

Lemma 5 *(a) is not possible, that is, there is no directed cycle in the final network scenario.*

Proof:
Suppose there is a cycle in the final graph. This cycle has finitely many events, so all these events must have been added by some finite stage n. Then, using lemma 1, the cycle is in the graph at stage n, contradicting the lemma that the graph at every finite stage is acyclic. ∎

Lemma 6 *(b) is not possible, that is, there is no infinite descending chain.*

Proof:
Suppose there is an infinite descending chain $e_0 - e_1 - \ldots$ in the final graph. e_0 was added at some finite stage n. Using lemma 1, the entire chain must be in the graph at stage n, contradicting the lemma that the graph is well-founded at every finite stage. ∎

Theorem 1 *A network composed of at most countably many monotone and continuous components is monotone.*

Proof:
By the described construction and the above lemmas. ∎

This by itself is not quite enough since the definition of implementation is stated in terms of the input-output relation. We need to know that restricting a channel does not destroy monotonicity.

Lemma 7 *If N is a monotone network, and a new network N' is obtained from N by hiding some input channels and some output channels of N, then N' is monotone.*

Proof:
Suppose S'_1 is a scenario for N', and suppose that S_1 is the scenario for N from which S'_1 was obtained. We extend the input histories in S'_1, thus extending the input histories in S_1. By monotonicity of N, we get a scenario S_2 with the extended input histories, such that $S_1 \sqsubseteq S_2$. Let S'_2 be the corresponding scenario for N'. We need to show that $S'_1 \sqsubseteq S'_2$.

First, for every input or output channel of N', the history in S'_1 is the same as the history in S_1. Since $S_1 \sqsubseteq S_2$, the history in S_1 is a prefix of the history in S_2, and this is equal to the history in S'_2. Hence, for every input or output channel of N', the history in S'_1 is a prefix of the history in S'_2.

Next, we need to show that there does not exist $\langle u,v\rangle \in S'_2$, such that $\langle u,v\rangle \notin S'_1$, and $v \in$ vertex-set(S'_1). Suppose $\langle u,v\rangle \in S'_2$ and $\langle u,v\rangle \notin S'_1$ and $v \in$ vertex-set(S'_1) $\subseteq$ vertex-set(S_1). Then $\langle u,v\rangle \in S_2$, and this implies that $\langle u,v\rangle \in S_1$, because $S_1 \sqsubseteq S_2$ and $v \in$ vertex-set(S_1). Since $\langle u,v\rangle \in S'_2$, neither u nor v corresponds to a hidden channel. Hence $\langle u,v\rangle \in S'_1$, which contradicts the assumption.

Next, we need to show that for any two vertices u and v in vertex-set(S'_1), the pair $\langle u,v\rangle \in S'_1$ iff $\langle u,v\rangle \in S'_2$. This is clear by using this fact for S_1 and S_2, since $S_1 \sqsubseteq S_2$. ∎

Corollary 1 *Any network built up from monotone and continuous networks is monotone.*

Proof:
Any network built up from monotone and continuous networks is obtained by composing the networks and then hiding some input and output channels. As proved earlier, any network obtained by composing monotone and continuous components is monotone. By the earlier lemma, hiding some input and output channels preserves monotonicity. Hence the resulting network is monotone. ∎

Lemma 8 **Amb** *and countably nondeterminate processes are monotonic and continuous.*

Proof:
A straightforward examination of the possible scenarios for the processes. ∎

Lemma 9 *The IO-relation of* **fair merge** *is not monotone.*

Proof:
Let the input channels of the **fair merge** be a and b and the output channel be c. Consider a computation where the input on channel a is the single token 1 and the input on b is the stream 2^∞. The possible outputs on c are of the form $2^i \cdot 1 \cdot 2^\infty$. Now consider a computation where we extend the input on a to $1 \cdot 1$. The possible outputs in this case are of the form $2^i \cdot 1 \cdot 2^j \cdot 1 \cdot 2^\infty$. None of these are extensions of any of the former output streams. Thus the IO-relation of **fair merge** is not monotonic. ∎

Corollary 2 *A* **fair merge** *cannot be implemented by any finite network of recursively defined M-processes and copies of processes in M, where M is the set of all deterministic Kahn processes,* **amb** *and countably nondeterminate (CN) processes.*

Proof:
Suppose N is any finite network of recursively defined M-processes and copies of processes in M, where M is the set of all deterministic Kahn processes, **amb** and CN processes. By earlier lemmas, all deterministic Kahn processes, **amb** and CN processes are monotone and continuous. So, by corollary 1, the completely expanded network for N is monotone. Hence N is monotone, because N and its completely expanded form have the same scenario sets. But fair merge is not monotone, by an earlier lemma. So no such network N can implement fair merge. ∎

4 Conclusions

The results in this paper establish that fair merge is a powerful primitive that does more than merely cause countable non-determinism. Past investigations have focussed on differentiating countable non-determinism from finite non-determinism. We believe that we have uncovered the first of a whole class of such results. Other investigations that we have carried out show that signaling adds to the expressive power of dataflow networks. Thus there are more levels of expressiveness then had been suspected before. We are working on consolidating these results into a systematic hierarchy of non-deterministic primitives. It is also an interesting question to investigate how these expressiveness properties relate in the presence of richer languages, for example, those with λ-abstraction or higher-order functions.

It is necessary to describe an operational semantics corresponding to the denotational model presented here to formally justify the reasonableness of the model. We have not presented an operational semantics here, but the work of Stark [24] shows that satisfactory operational semantics can be defined. We are working on proving the equivalence between the operational semantics of Stark and the scenario model.

A related area of investigation into expressiveness is the finite delay operators of Milner's Calculus for Communicating Systems (CCS) [13]. One finds in the literature on CCS two different primitives for expressing finite delay, one due to Milner [14] and one due to Hennessy[6]. Milner's finite delay operator will always produce finite delay sequences whereas Hennessy's will produce an infinite delay sequence if there is nothing it can synchronize with but only finite delay if there is something for it to synchronize with. Preliminary work that we have done shows that these are provably inequivalent. This shows that our results are not due to some idiosyncrasy of asynchronous systems.

5 Acknowledgements

We would like to thank Radhakrishnan Jagadeesan for extremely useful discussions. We would also like to thank Abha Moitra for reading an earlier draft of this paper. We have also benefited from discussions with Gene Stark, Elsa Gunter, Glynn Winskel, Joachim Parrow and Samson Abramsky. This research was supported by NSF grant DCR-8602072 to Cornell University.

References

[1] Samson Abramsky. On semantic foundations for applicative multiprogramming. In J. Diaz, editor, *Proceedings of the Tenth International Conference On Automata, Languages And Programming*, pages 1–14, New York, 1983. Springer-Verlag.

[2] Samson Abramsky. Private communication, 1984.

[3] K. R. Apt and Gordon Plotkin. Countable nondeterminism and random assignment. *Journal Of The ACM*, 33(4):724–767, 1986.

[4] R. J. Back. A continuous semantics for unbounded non-determinism. *Theoretical Computer Science*, 23(2):187–210, 1983.

[5] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, pages 252–259, 1981. LNCS 107.

[6] Mathew Hennessy. Axiomatizing finite delay operators. *Acta. Inform.*, pages 61–88, 1984.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.

[8] Robert M. Keller. Denotational models for parallel programs with indeterminate operators. In *Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[9] Robert M. Keller and Prakash Panangaden. Semantics of networks containing indeterminate operators. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 479–496, 1985. LNCS 197.

[10] Robert M. Keller and Prakash Panangaden. Semantics of digital networks containing indeterminate operators. *Distributed Computing*, 1(4):235–245, 1986.

[11] Joost Kok. Denotational semantics of nets with nondeterminism. In *Proceedings of the 1986 European Symposium on Programming*, pages 237–249, 1986.

[12] Joost Kok. A fully abstract semantics for dataflow nets. In *Proceedings of Parallel Architectures And Languages Europe 1987*, pages 351–368, Berlin, 1987. Springer-Verlag.

[13] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[14] Robin Milner. A finite-delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1982.

[15] Prakash Panangaden. Abstract interpretation and indeterminacy. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 497–511, 1985. LNCS 197.

[16] Prakash Panangaden and Vasant Shanbhogue. On the expressive power of indeterminate primitives. Technical Report 87-891, Cornell University, Computer Science Department, November 1987.

[17] Prakash Panangaden and Eugene W. Stark. Computations, residuals and the power of indeterminacy. In *Proceedings of the Fifteenth ICALP*. Springer-Verlag, 1988.

[18] David Park. On the semantics of fair parallelism. In *Proceedings of the Winter School on Formal Software Specification*, pages 504–526, New York, 1980. Springer-Verlag. Lecture Notes In Computer Science 86.

[19] David Park. The "fairness problem" and non-deterministic computing networks. In *Proceedings of the Fourth Advanced Course on Theoretical Computer Science, Mathematisch Centrum*, pages 133–161, 1982.

[20] Gordon Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[21] Gordon Plotkin. Lcf considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[22] Vaughn Pratt. On the composition of processes. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 213–223, 1982.

[23] Vaughn Pratt. Modeling concurrency with partial orders. *International Journal Of Parallel Programming*, 15(1):33–71, 1986.

[24] Eugene W. Stark. Concurrent transition system semantics of process networks. In *Proceedings Of The Fourteenth Annual ACM Symposium On Principles Of Programming Languages*, pages 199–210, 1987.

[25] Glynn Winskel. Event structures. Technical Report 95, University of Cambridge, Computer Laboratory, 1986.