On The Expressive Power of Indeterminate Network Primitives

Prakash Panangaden
Vasant Shanbhogue

87-891
December 1987

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

On The Expressive Power of Indeterminate Network Primitives

Prakash Panangaden
and
Vasant Shanbhogue
Computer Science Department, Cornell University

December 12, 1987

Abstract

It is well known that a fair merge primitive leads to unbounded indeterminacy. In this paper we show that unbounded indeterminacy cannot express a fair merge in the setting of Kahn-style dataflow networks. Intuitively, unbounded indeterminacy can be used to program a fair merge when it is guaranteed that data will always be available. But such schemes rely on predictive scheduling and they may fail if one of the inputs to the merge is a finite stream. It is reasonable to expect that if one were to add a primitive which "knows how to avoid bottom" (so called "angelic merge") then one could use this in conjunction with unbounded choice in order to produce a fair merge. Somewhat surprisingly, this expectation is incorrect as we show in this paper. The method we use to prove this is to identify a property which generalises monotonicity to indeterminate networks and then show that this property is possessed by determinate networks and by unbounded choice and by angelic merge but not by fair merge. It appears that there is a hierarchy of inequivalent indeterminate primitives all of which feature some form of unbounded indeterminacy.

1 Introduction

In this paper we examine the relative expressive power of various indeterminate primitives in the context of static dataflow networks. Specifically, we study the relationship between *fair merge* and primitives which feature unbounded choice. An example due to Park shows that using a certain primitive called *amb* and recursion one can get unbounded indeterminacy but it was not known how to get fair merge using the same primitives. That example relies on the ability to expand recursive definitions and obtain new copies of a function. The semantics of dynamic indeterminate networks is, in our opinion, still not fully resolved. In particular, there is no satisfactory treatment that we know of which relates the denotational semantics to an operational semantics. In fact, Meyer points out that one can argue that Park's example does not really exhibit indeterminacy but his argument is based on a sequential operational semantics [10].

In an earlier paper by Panangaden and Stark [15] the relative expressive power of fair merge and angelic merge is established. These proofs use an operational semantics. In the present paper, we establish the same results using a denotational semantics and also show that angelic merge cannot be implemented by unbounded choice. We believe this approach makes it clearer to understand the nondeterminism issues and may be generalised for the purpose of studying other such issues.

In order to clarify the issues, therefore, we have decided to look at the question of what it takes to implement a fair merge with static dataflow networks. This study will of course not settle the corresponding questions for dynamic networks but we will show that there are surprises even at this level. In particular we show that unbounded indeterminacy and the ability to detect and avoid nonterminating computations even combined do not suffice to implement a fair merge.

Our picture of a network is essentially the same as a Kahn network [5] with indeterminate primitives added. Thus we view a network as a static graph whose arcs are unbounded queues and whose vertices are autonomous computing agents; we shall call these computing agents *nodes*. The nodes communicate with each other by passing messages along the arcs. The determinate nodes are allowed to communicate with the other nodes only by reading from or writing to the channels. Some of the indeterminate

nodes may test their input channels for availability of data and branch on the result. In this paper, we argue that this ability to branch on availability of data is essential if one wants to implement a fair merge. A primitive with this property was proposed by Keller [6] and called by him *poll*. He showed a simple program which uses poll to implement fair merge. We consider a suitable abstraction of poll and show that it can implement fair merge and be implemented by fair merge. Our expressiveness theorems will be expressed in terms of poll rather than fair merge because poll is a bit simpler to describe.

The next section sketches the semantic model and makes precise what we mean by "implements". The following sections introduce the property which generalises monotonicity and uses this to prove the expressiveness result. We conclude with a discussion of related issues and some new expressiveness conjectures.

2 The Semantic Model

The semantic model we use is essentially that of Keller and Panangaden [8, 14,7]. In this section we summarise this model. We will not need the fixed-point formulation of the semantic theory, all we need is a compositional theory so that we can prove semantic properties of networks by induction on the structure of networks.

As with most of the other proposed models of networks with indeterminate nodes, our basic semantic unit is an *event* [2,3,18,13,20,9,19].

Definition 1 *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* $\mathbf{a}$.

Sometimes we need to distinguish the appearance of a token on a channel from its consumption. In this case we shall define events to be triples with an extra tag $\pm$ indicating whether the event corresponded to the appearance or the consumption of the token on the indicated channel. Thus the event $(\mathbf{a}, +, v)$ means that the value v appeared on channel a. Similarly, the event $(\mathbf{a}, -, v)$ is interpreted as the reading or consumption of the value v from channel $\mathbf{a}$. Using these triples leads to a finer notion of network

equivalence since we can indicate that a token appeared on a channel and was not consumed. For the purposes of proving the expressiveness results the weaker notion suffices. Indeed the theorem is stronger if we use a weaker notion of network equivalence. The notion that we use is even weaker then conventional trace equivalence.

Definition 2 *A* trace *is a sequence of events that happen on the various channels of a network.*

A possible execution or run of a network is described by a trace; the order of events in a trace is the temporal order in which the events occurred in that particular execution of the network.

Definition 3 *The set of possible traces of a network is called its* archive *and defines the meaning of a network.*

In order to specify archives conveniently it is useful to define various operations and functions on traces. We shall use the notation $\Pi_b(x)$ to stand for the sequence of events from the trace x that occur on channel **b**. Thus Π can be thought of as a projection operator on traces. We shall use the notation $\Pi_{b=v}(x)$ for the sequence obtained from x by projection on those **b**-events for which the value is v and $\Pi_B(x)$ for the sequence obtained by projecting onto all the members of the set of channels B. Dual to Π we use the operator K to stand for removal of events of a particlar form from a trace. Thus, for example, we would write $K_b(x)$ for the trace obtained from x by deleting all the events which occurred on the channel **b**. We shall use the symbol $\sqsubseteq$ to stand for the prefix relation between sequences and the symbol $\#$ to stand for the number of elements in a sequence. If the sequence x is infinite then $\#(x)$ is undefined.

Certain predicates on traces are also useful enough to need their own notation. We use the predicate "precedes" to indicate temporal precedence between events in a trace. Thus $precedes(\mathbf{b}, \mathbf{c}, x)$ means that on the trace x every **c**-event is preceded by an identical **b**-event. More precisely,

Definition 4 *We say that* $precedes(\mathbf{b}, \mathbf{c}, x)$ *holds if, for every prefix* $y \sqsubseteq x$ *we have* $\#(\Pi_c(y)) \leq \#(\Pi_b(y))$ *and for every prefix* $y \sqsubseteq x$ *there is another prefix* $z \sqsubseteq y$ *such that* $\Pi_c(y) = \Pi_b(z)$.

This is useful for defining the archives corresponding to networks with feedback loops. Finally we use the simple predicate "equal" to indicate that the sequences of values on two channels are the same in a particular trace. Formally, $equal(\mathbf{b}, \mathbf{c}, x)$ means that the sequences of values on channels **b** and **c** are identical.

We can now define the archive for an unbounded buffer with input channel **a** and output channel **b** as follows:

$$\{x | equal(\mathbf{a}, \mathbf{b}, x) \wedge precedes(\mathbf{a}, \mathbf{b}, x)\}$$

We now describe network composition. There are three ways in which networks can be composed. These are aggregation, loop formation and serial composition. (Note that network composition does not hide any channels.)

In aggregation, two networks are viewed as a single network but no channels are connected. The notation for this is $N \| M$. The corresponding construction on archives is shuffling. We use the symbol Δ to stand for the fair shuffle operator. Given two sets of sequences X and Y, the set $X \Delta Y$ is obtained by shuffling every sequence in X with every sequence in Y.

Definition 5 *The archive for the aggregate of two networks is given by the shuffle operator;*

$$Archive(N \| M) = Archive(N) \Delta Archive(M)$$

.

The formation of feedback loops requires more care. The syntax we use for the network obtained from network N by connecting the output channel b to the input channel c is $loop(\mathbf{b}, \mathbf{c}, N)$. The corresponding operation on archives is given by:

Definition 6 *The archive for* $loop(\mathbf{b}, \mathbf{c}, N)$ *is given by*

$$Archive(loop(\mathbf{b}, \mathbf{c}, N)) = \{K_c(x) | x \in Archive(N) \wedge precedes(\mathbf{b}, \mathbf{c}, x)\}$$

This expresses the basic idea that in the looped network the events on **c** arise from events on **b**. With these two definitions in place it is clear that

sequential composition of two networks can be viewed as an aggregation followed by a loop composition. The resulting construction on archives is easy to derive; see [8,7] for details.

An important property that all archives are required to satisfy is called *input freeness*. Intuitively, input-freeness expresses the idea that the description of an archive cannot put constraints on the values and relative temporal order of events on the input channels to a network. We shall use this property frequently so we shall define it in detail. This property was not discussed in earlier presentations of this semantic model. Let $\mathcal{A}$ be any fixed archive. Suppose h is a trace from $\mathcal{A}$. Suppose that **a** is an input channel to the network which has $\mathcal{A}$ as its archive. Suppose that $\Pi_a(h) = \alpha$. Then it should be possible to have a new input event on channel **a** added arbitrarily later and have a trace in $\mathcal{A}$ which has this extended input.

Definition 7 *We say that $\mathcal{A}$ satisfies* input-freeness *if the following condition is satisfied. Suppose that $\alpha' = \alpha.e$ where e is a new event on* **a** *which occurs after all the events on α. Then for every prefix $\overline{h} \sqsubseteq h$ such that $\Pi_a(\overline{h}) = \alpha$ there is a trace $h' \in \mathcal{A}$ such that $\overline{h} \sqsubseteq h'$ and $\Pi_a(h') = \alpha'$ and if the length of $\overline{h}$ is k then the $(k+1)st$ member of h' is e.*

Thus we can always extend the input sequence and find some trace which has that as its input sequence. We, of course, do not require that the network produce any output to respond to this input. By introducing the new event as late as we please we can ensure that any new response to this new input event will occur after the old output has occurred unless the old output was already infinite.

The final issue to be taken care of in this section is the notion of implementation which we use in our expressiveness theorems. Roughly speaking, we say one network implements another if they compute the same input-output relation. To make this precise we define the *graph of an archive* first.

Definition 8 *Let $\mathcal{A}$ be the archive of a network N. Let the input channels to $\mathcal{A}$ be $\vec{\alpha}$ and the output channels be $\vec{\beta}$. Then the graph of the archive $\mathcal{A}$ is a set of pairs of vectors of sequences given by*

$$\mathcal{G}(\mathcal{A}) = \{< \vec{\alpha}, \vec{\beta} > | \Pi_{\vec{a}}(h) = \vec{\alpha} \wedge \Pi_{\vec{b}}(h) = \vec{\beta} \text{ where } h \in \mathcal{A}\}$$

We shall say that a network N *implements* network M if they have the same graph upto renaming of the channels. By this definition, we do not care if the two networks have different internal configurations and thus, different archives, but they must present the same interface to the external world.

3 Network Monotonicity and Continuity

In this section we define network monotonicity and network continuity. It is difficult to prove that network monotonicity is preserved under network composition (which is what we want). So we prove that a stronger condition, network monotonicity and continuity, is preserved under network composition.

The intuition behind the definition is that a monotonic network is one in which if the enabling conditions for a particular computation exist then the arrival of data cannot disable that possible computation. This is clearly a property that nodes that have time sensitivity do not possess. It is also intuitively clear that a network composed of such subnetworks must also possess this property. However, in order to prove that this property is preserved by loop composition we need to introduce an analogue of continuity and show that continuity is preserved by network composition.

In order to define monotonicity and continuity, we need to define the complete graph of a network. This generalizes the definition of graph by referring to all the channels of a network rather than just the input and output channels.

Definition 9 *The complete graph, $\mathcal{C}(N)$ of a network, N, is*

$$\mathcal{C}(N) = \{\langle \Pi_{a_1}(x), \Pi_{a_2}(x), \ldots, \Pi_{a_n}(x)\rangle \mid x \in Archive(N)\}$$

where, the channels of N are $\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}$.

One needs to consider the complete graph, rather than just the graph, when one is working with nondeterminate networks.

Definition 10 *Let N be a network with input channels $\vec{a}$. Let the rest of the channels be $\vec{b}$ and the archive of N be $\mathcal{A}$. Let $(\vec{\alpha}, \vec{\beta})$ be in $\mathcal{C}(\mathcal{A})$. Let h be a trace in $\mathcal{A}$ such that $\Pi_{\vec{a}}(h) = \vec{\alpha}$ and $\Pi_{\vec{b}}(h) = \vec{\beta}$. Suppose $\vec{\alpha} \sqsubseteq \vec{\alpha}'$ where*

the symbol $\sqsubseteq$ is extended componentwise to vectors in the obvious way. We say that the network N is monotonic *if, for every prefix, $\overline{h}$ of h such that $\Pi_{\vec{a}}(\overline{h}) = \vec{\alpha}$ there is a trace h' in $\mathcal{A}$ with $\overline{h} \sqsubseteq h'$ and with $\Pi_{\vec{a}}(h') = \vec{\alpha}'$ and with $\vec{\beta} \sqsubseteq \Pi_{\vec{b}}(h')$ and with the relative ordering of the channel events in h preserved in h'.*

This says that we can add new input arbitrarily late in an execution (hence the quantification over prefixes of h) and there will be some trace in the archive which represents the response of the network to the new input and in this new trace, the sequence of tokens on each channel will be an extension of the sequence of tokens seen before. Clearly we cannot expect that every response will be an extension of the old response since the networks are indeterminate. This captures the idea that if there was an enabled output then adding new input will not disable this output. Note that this definition of monotonicity implies that the output channels will have extended sequences as well.

Now we consider the properties of determinate primitives. By a *primitive* we mean a network with no internal channels. One can think of a primitive as a sequential process communicating with the rest of the network through its input and output channels. For a network primitive the notions of graph and complete graph coincide.

Definition 11 *We say that a network primitive is determinate if the graph of the archive is functional.*

Theorem 1 *The graph computed by a determinate network is monotonic.*

Proof:
Suppose that $\mathcal{A}$ is the archive for a determinate network N. Let $(\vec{\alpha}, \vec{\beta})$ be in $\mathcal{G}(\mathcal{A})$. Suppose that $\vec{\alpha} \sqsubseteq \vec{\alpha}'$. By input-freeness, we must have some $(\vec{\alpha}', \vec{\beta}')$ in $\mathcal{G}(\mathcal{A})$. By determinacy, we know that there is only one pair in $\mathcal{G}(\mathcal{A})$ which has $\vec{\alpha}'$ as its first component. Suppose that $\vec{\beta} \not\sqsubseteq \vec{\beta}'$. Then, for some $i \in \{1, \ldots, n\}$ $\beta_i \not\sqsubseteq \beta_i'$. This means that there is some point in the sequences at which they first differ, let us assume that this happens at the kth position in the sequence and that the value here is v in β, and not v in β'. Now consider corresponding traces h, h' for $(\vec{\alpha}, \vec{\beta})$ and $(\vec{\alpha}', \vec{\beta}')$ respectively. These need not be unique. In the trace h there is an event

$(\mathbf{b_i}, v)$ at the kth position in the sequence for channel b_i. This event does not occur at the kth position in the sequence for channel b_i in h'. By input-freeness there must be a trace which has the new events from $\vec{\alpha'}$ inserted after any prefix of h. In particular we must have a trace which extends a prefix of h that includes $(\mathbf{b_i}, v)$ and which has a projection onto its input channels which equals $\vec{\alpha'}$. This trace cannot possibly have a projection onto its output channels which equals $\vec{\beta'}$ since we know β'_i at its kth position does not have the event $(\mathbf{b_i}, v)$. But now the graph is no longer functional. Thus we must have $\vec{\beta} \sqsubseteq \vec{\beta'}$. $\Box$

The intuition here should be clear, if the arrival of new inputs disables previously enabled events, then the network is sensitive to the arrival times of data and cannot possibly be determinate, since the graph does not express any timing information.

The next step is to show that the composition of networks preserves monotonicity . There are two different forms of network composition that we need to consider namely aggregation and loop composition. Of these, aggregation is more or less trivial whereas loop composition, not surprisingly, involves an inductive proof. In order for the inductive proof to go through, we need to prove that a property stronger than monotonicity is preserved by network composition.

Definition 12 *A network, N, is said to be* **continuous** *if its complete graph satisfies the following property. Suppose that $\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n}, \ldots$ is a sequence of tuples from $\mathcal{C}(N)$ such that $\forall i, j \in Nat.\ i < j \Rightarrow \vec{v_i} \sqsubseteq \vec{v_j}$, where $\sqsubseteq$ is the prefix order on sequences extended pointwise to vectors of sequences. Then $\vec{v}$ must be in $\mathcal{C}(N)$, where $\vec{v}$ is the least upper bound of the sequence of vectors computed componentwise.*

Gilles Kahn showed that [5] determinate primitives were continuous in this sense. We will show that a network composed of monotonic and continuous components has to be monotonic and continuous.

Theorem 2 *The aggregate of two monotonic and continuous networks, N and M, is monotonic and continuous.*

Proof:

Let M and N be monotonic and continuous networks and their archives be $\mathcal{A}$ and $\mathcal{B}$ respectively. Let $\mathbf{a_1}, \ldots, \mathbf{a_n}$ be the input channels of N and let $\mathbf{c_1}, \ldots, \mathbf{c_m}$ be the input channels of M. Suppose that $\langle \sigma_1, \ldots, \sigma_k \rangle$ is in $\mathcal{C}(N||M)$. Then, because the channels of N and M are disjoint, we can decompose this into $\langle \xi_1, \ldots, \xi_i \rangle$ and $\langle \zeta_1, \ldots, \zeta_j \rangle$ where the ξs are sequences on the channels of N and the ζs are sequences on the channels of N. Let the vector of input sequences to N be $\langle \alpha_1, \ldots, \alpha_n \rangle$ and let the vector of input sequences to M be $\langle \beta_1, \ldots, \beta_m \rangle$. We shall write $\langle \gamma_1, \ldots, \gamma_{n+m} \rangle$ for the vector of input sequences to the aggregate network $N||M$. In order to establish monotonicity of the aggregate, we need to show that if the vector of γs is extended then there is a trace such that when projected onto the input channels we get the extended inputs and when projected onto the other channels we get extensions on all of them. Let the extended input sequences be $\langle \gamma'_1, \ldots, \gamma'_{n+m} \rangle$. Then this can be decomposed into $\langle \alpha'_1, \ldots, \alpha'_n \rangle$, input sequences for N, and $\langle \beta'_1, \ldots, \beta'_m \rangle$ input sequences for M. Clearly, the vector of primed αs extends the original sequence of αs and similarly for the βs. Since N and M are both monotonic, there are traces h_N and h_M, of N and M respectively, such that when projected onto their input channels yield $\langle \alpha'_1, \ldots, \alpha'_n \rangle$ and $\langle \beta'_1, \ldots, \beta'_m \rangle$ respectively, and when projected onto all their channels yield extensions of $\langle \xi_1, \ldots, \xi_i \rangle$ and $\langle \zeta_1, \ldots, \zeta_j \rangle$ respectively. The archive of the aggregate network contains $h_N \Delta h_M$ which, when projected onto its input channels gives $\langle \gamma'_1, \ldots, \gamma'_{n+m} \rangle$ and when projected onto all its channels, gives an extension of $\langle \sigma_1, \ldots, \sigma_k \rangle$.

The proof that continuity is preserved is very similar. One can decompose elements of the complete graph of $N||M$ to obtain elements of the complete graphs of N and M. One can obtain the required limiting vector by using the continuity of the components and shuffling the corresponding traces. □

Now we consider the case of loop composition. For definiteness, let us suppose that the network in question is M and that the output channel $\mathbf{b}$ of M is connected to the input channel $\mathbf{c}$ forming a feedback loop. Let the archive for M without the loop be $\mathcal{A}$ while the archive for the network with the loop is $\mathcal{B}$. Recall that the relationship between $\mathcal{A}$ and $\mathcal{B}$ is

$$\mathcal{B} = \{K_c(x) | x \in \mathcal{A} \wedge precedes(\mathbf{b}, \mathbf{c}, x)\}.$$

We will use the following convenient notation. If $\vec{\gamma}$ is in the complete

graph of a network and suppose that the input sequences of $\vec{\gamma}$ are $\vec{\alpha}$ and the rest of the sequences are $\vec{\beta}$ then we will say that $(\vec{\alpha}, \vec{\beta})$ is in the complete graph of the network. This notation suggests an input-output relation but note that we include the sequences on the internal channels of the network in $\vec{\beta}$. Also, by $\vec{\alpha}; \gamma$ we will mean the vector of sequences consisting of sequences in $\vec{\alpha}$ and the sequence γ.

We will apply the adjectives 'monotonic' and 'continuous' to archives as well as networks.

Theorem 3 *Suppose the archives $\mathcal{A}$ and $\mathcal{B}$ are defined as above. If $\mathcal{A}$ is monotonic and continuous then $\mathcal{B}$ is monotonic and continuous also.*

Suppose that $(\vec{\alpha}, \vec{\beta})$ is in the graph of $\mathcal{B}$ and suppose that h is a trace that satisfies $\Pi_I(h) = \vec{\alpha}$ and $\Pi_C(h) = \vec{\beta}$, where $\mathbf{I}$ and $\mathbf{C}$ are the sets of input channels and rest of the channels, respectively, of the network with the loop. Now we consider extending the input to $\vec{\alpha}'$ where $\vec{\alpha} \sqsubseteq \vec{\alpha}'$. There is some trace h_0 of $\mathcal{A}$ such that $\Pi_I(h_0) = \vec{\alpha}; \Pi_c(h_0)$ and $\Pi_C(h_0) = \vec{\beta}$. Since the original network M is monotonic, we can extend the inputs to $\vec{\alpha}'; \Pi_c(h_0)$ and there will be some new trace of M, call it h_1, such that $\Pi_I(h_1) = \vec{\alpha}'; \Pi_c(h_0)$ and $\Pi_C(h_1) = \vec{\beta_1}$ with $\vec{\beta} \sqsubseteq \vec{\beta_1}$ and with the relative ordering of events from h_0 preserved in h_1. Now, however, we have no guarantee that h_1 satisfies the condition $precedes(\mathbf{b}, \mathbf{c}, h_1)$. In the rest of the discussion we describe how to build a trace, call it h', in $\mathcal{A}$ such that $\Pi_C(h') = \vec{\beta}'$, $\Pi_I(h') = \vec{\alpha}'$ and with $\vec{\beta} \sqsubseteq \vec{\beta}'$ which also satisfies $precedes(\mathbf{b}, \mathbf{c}, h')$. $K_c(h')$ will then be present in the archive $\mathcal{B}$ and will demonstrate that the the network with the loop is also monotonic.

The construction of h' will be carried out using traces from $\mathcal{A}$. Since we know that $\mathcal{A}$ is monotonic we can construct h' in successive approximations knowing that at each stage we can do it in such a way that the sequences of events on the channels (the β's) increase at each stage of the construction. Now since $\mathbf{c}$ is an input channel of M we can introduce (by input freeness) new $\mathbf{c}$ events which match the new $\mathbf{b}$ events in h_1, thus extending the input and getting a new trace h_2 (by monotonicity) in which the output has been extended.

We iterate this process and obtain, in this way, a sequence of traces $h_0, h_1 \ldots h_n \ldots$ such that the event sequences on all the channels are increasing and for which $\Pi_b(h_n) = \Pi_c(h_{n+1})$. At any stage, if we find that

$\Pi_b(h_n) = \Pi_b(h_{n+1})$ we know that we have produced a trace with the required property. In particular, if $\Pi_b(h_n)$ is infinite for any n, then $\Pi_b(h_n) = \Pi_b(h_{n+1})$. However, it is possible that when we look at the sequence of traces projected onto **b** we obtain an infinite sequence of finite sequences which are always properly increasing under the prefix ordering on sequences. In this case, we have a sequence of inputs $\vec{\alpha'}; \Pi_c(h_i)$ and a corresponding sequence of outputs $\vec{\beta_i}$. By the continuity of M, we have a trace h_∞ with input $\sqcup(\vec{\alpha'}; \Pi_c(h_i)) = \vec{\alpha'}; \sqcup\Pi_c(h_i)$ and outputs $\sqcup\vec{\beta_i}$ with the relative order of events from each h_n being preserved. The sequence corresponding to channel **b** is $\sqcup\Pi_b(h_i)$ in $\sqcup\vec{\beta_i}$.

Since $\Pi_b(h_n) = \Pi_c(h_{n+1})$, we have $\cup\Pi_b(h_i) = \sqcup\Pi_c(h_{i+1}) = \sqcup\Pi_c(h_i)$. Any event in $\sqcup\Pi_c(h_i)$ is in some $\Pi_c(h_i)$, hence preceded by a corresponding b-event in the trace h_i, and hence preceded by a corresponding b-event in $\cup\Pi_b(h_i)$ (because relative order is preserved). Therefore, $precedes(\mathbf{b}, \mathbf{c}, h_\infty)$ holds and so h_∞ is the required trace with input $\vec{\alpha'}$ and the rest of the sequences an extension of $\vec{\beta}$ with relative order of events preserved.

Now we prove that $\mathcal{B}$ is continuous. Suppose $(\vec{\alpha_1}, \vec{\beta_1}), \ldots \epsilon\ \mathcal{G}(\mathcal{B})$ with the corresponding traces being $h_1, \ldots$, then $(\vec{\alpha_1}; \Pi_c(h_1), \vec{\beta_1}), \ldots \epsilon\ \mathcal{G}(\mathcal{A})$. By continuity of $\mathcal{A}$, $\exists$ a trace h with input $\sqcup\vec{\alpha_i}; \Pi_c(h_i)$ and rest of the sequences $\sqcup\vec{\beta_i}$ with relative order of events in every h_i being preserved. Since $\Pi_c(h_i) = \Pi_b(h_i)$ for every i, $\sqcup\Pi_c(h_i) = \sqcup\Pi_b(h_i)$ and as before, $precedes(\mathbf{b}, \mathbf{c}, h)$ holds. So $K_c(h)$ is a trace for $\mathcal{B}$ with the required property. □

4 The Expressiveness Theorem

In this section, we prove the expressiveness theorem by showing that fair merge is not monotonic, but that angelic merge and unbounded choice are monotonic. An angelic merge node has two input channels, **a**, **b** and an output channel **c**. Informally we may describe it as follows; if the input sequence on **a** is finite then every value that appears on **b** is read and written on **c** and vice-versa. If both input sequences are finite then the angelic merge acts fairly while if both are infinite it may not act fairly in choosing values from the two input channels but it will produce infinite output. Thus an angelic merge "knows how to avoid bottom" but does not know how to be fair on infinite streams. The archive for angelic merge

is tedious to write down formally but the possible traces should be clear. Consider the situation when the input sequence on one of the channels is finite. Extend the sequence. If the other input channel also has a finite sequence on it then there will be a trace in which the new output values appear after all the former output and we get an extension of the previous output. If the other channel is infinite then there will be a trace in which the new input values are ignored and the new output equals the old output. If the input sequences are both infinite then they cannot be extended.

An *iterated unbounded choice* node has one input channel **r** and one output channel **s**. It consumes any value on the input channel and produces any positive integer on the output channel. In particular, we could feed it an infinite stream of zeros and have it produce an infinite stream of arbitrary positive integers on its output channel. This node features unbounded nondeterminism. With iterated unbounded choice (or simply "choice" henceforth) we can easily program a merge node which is fair if both its input streams are infinite. We use the choice node to produce an arbitrary stream of integers. We can then use this as an "oracle" for a determinate node which has three inputs and a single output. The determinate node uses the arbitrary stream of integers to determine how many tokens to read off each of its input channels before switching to the other input channel. This is called an *infinity-fair* node by Park [16]. This trick is well known, see, for example, Apt and Plotkin [1]. One would expect that with angelic merge and iterated unbounded choice together one should be able to implement a fair merge but this turns out to be impossible. The archive here is simply:

$$\{x | \#(\Pi_r(x)) = \#(\Pi_s(x)) \text{ and } precedes(\mathbf{r}, \mathbf{s}, x)$$

The input values provide no constraint on the output values, there has to be exactly one output for each input. It is trivial to check that this is monotonic.

It turns out that fair merge requires, informally speaking, time sensitivity. Suppose we were to add to Kahn's original language [5] a primitive called *poll*[6]. One may think of poll expressions as having the syntax **poll a** where **a** is the name of an input channel. This returns *false* if there is no value available to read on channel **a** and *true* if there is. It is easy to program a fair merge, one simply cycles forever polling the two input channels

and reading data if it is present. This node never waits to read data that is not going to appear nor will it consistently favour a particular channel. We can abstract away from the low-level consideration and introduce a new network primitive called poll which has "essentially" the same behaviour. A poll primitive has one input channel **p** and one output channel **q**. Informally every input value is read and copied onto the output and in between any two such output values there is a finite sequence of special values say $\star$s. By special value we mean one that cannot be produced by other nodes. If the input sequence is finite then the output sequence ends with an infinite sequence of $\star$ values whereas if the input sequence is infinite then the output sequence has finite blocks of $\star$s between every two ordinary values. This node is intended to capture the idea that we can test for the availability of data and signal the result. Note that every trace for poll is infinite and indeed has infinitely many occurrences of $\star$ on its output. With poll one can easily implement fair merge. All one needs is a determinate zipper, i.e. a network which has two input channels and reads tokens from each one and copies them onto its output channel in strict alternation. To implement a fair merge one puts a poll in front of each input channel to the zipper. Then the resulting output can be filtered to remove the $\star$'s. It is trivial to implement poll with a fair merge, just merge the input stream with an infinite sequence of $\star$'s.

The formal description of the archive for poll is:

$$\{x | \Pi_{q=\star}(x) \text{is infinitely long} \wedge \forall y.(y \sqsubseteq x) \Rightarrow (\exists z.(y \sqsubseteq z \sqsubseteq x) \wedge \Pi_p(y) = \Pi_{q\neq\star}(z)\}$$

This condition ensures that every possible trace is infinite and that every value which appears on the input channel eventually appears on the output channel.

Lemma 1 *Any network obtained by hiding some non-input channels of a monotone and continuous network is monotone.*

Proof :
Let $\vec{\alpha}$ be the sequences on the input channels and $\vec{\beta}$ be the sequences on the rest of the channels. Let $\vec{\gamma}$ be the sequences on some of the non-input channels (when the rest of the non-input channels are hidden). When $\vec{\alpha}$ is extended to $\vec{\alpha'}$, then by monotonicity of the network, there is a trace

with input sequences $\vec{\alpha}'$ and rest of the sequences $\vec{\beta}'$, and $\vec{\beta} \sqsubseteq \vec{\beta}'$. But $\vec{\gamma}$ is contained among the $\vec{\beta}$. So $\vec{\gamma}$ also gets extended to some $\vec{\gamma}'$ in that trace. The trace, restricted to the non-hidden channels, is a trace of the new network. Hence the new network is monotonic. □

Corollary 1 *Any network implemented by monotone and continuous components is monotone.*

Proof :
Any network implemented by monotone and continuous components is obtained by composing the components and then hiding some non-input channels. As proved earlier, any network obtained by composing monotone and continuous components is monotone and continuous. By the earlier lemma, hiding some non-input channels preserves monotonicity. Hence the resulting network is monotone. □

Theorem 4 *Angelic merge primitives, iterated unbounded choice primitives, and any determinate primitives cannot together implement poll or fair merge.*

Proof :
By Corollary 1, any network implemented by the above is monotonic. But poll is not monotone. To see this, consider the case where there is only one value, say v, on the input channel. The possible traces projected onto the output channel are of the form $(\star)^i v(\star)^\infty$. Now consider extending the input to be v followed by w. Now the possible output sequences are of the form $(\star)^i v(\star)^j w(\star)^\infty$. None of these latter sequences are extensions of any of the former sequences. This proves that no network composed of any determinate primitives or angelic merge nodes or iterated unbounded choice nodes can possibly implement poll or fair merge. □

Note that, although fair merge is also not continuous, and continuity is preserved under composition, it might seem that we could prove the expressiveness theorem using the continuity property instead of monotonicity. This is not possible, because a new primitive, ∞-fair merge, that is equivalent (under implementation) to iterated unbounded choice, can be obtained by composing some monotone and continuous networks and then hiding some non-input channels. But this is not continuous.

An ∞-fair merge takes two input sequences. It can wait indefinitely for input from one input channel, but if one of the input sequences is infinite, then it must read the other input sequence completely. So, if both input sequences are infinite, then it reads both the input sequences completely and outputs a fair merge of the two sequences (hence the name ∞-fair merge). Now consider the inputs (1,2), (1;1, 2), (1;1;1, 2), $\cdots$. 1, 1;1, 1;1;1, $\cdots$ are possible outputs respectively. The limit of the inputs is $(1^\infty, 2)$ and the limit of the outputs is 1^∞, which is not a valid output because 2 has not been output.

So we cannot just consider continuity and need to consider monotonicity.

5 Iterated unbounded choice cannot implement angelic merge

In this section we show that iterated unbounded choice cannot implement angelic merge. This makes clear what we intuitively expect — even unbounded nondeterminism cannot implement a way of "avoiding bottom" on an input channel.

Theorem 5 *Iterated unbounded choice cannot implement angelic merge.*

Corollary 2 *No network which can be implemented by iterated choice can implement angelic merge.*

Proof : Define prefix closure property P as :

> If $(\vec{\alpha}, \vec{\beta_i})$, i = 1,2, $\cdots$, k are all the elements of $\mathcal{G}(\mathcal{A})$ having $\vec{\alpha}$ at the input channels, and if $\vec{\alpha} \sqsubseteq \vec{\alpha'}$, and $(\vec{\alpha'}, \vec{\beta}) \in \mathcal{G}(\mathcal{A})$, then $\exists i\ \vec{\beta'} \sqsupseteq \vec{\beta_i}$ and the order of events in $\vec{\alpha}$ and $\vec{\beta_i}$ is the same in the history corresponding to $(\vec{\alpha}, \vec{\beta_i})$ as in the history corresponding to $(\vec{\alpha'}, \vec{\beta'})$.

Informally, if an input I produces a particular output O, then any prefix of the input I produces some prefix of the output O as one possible output.

(a) Deterministic Kahn networks have P.
$(\vec{\alpha}, \vec{\beta}) \in \mathcal{G}(\mathcal{A})$ and $(\vec{\alpha'}, \vec{\beta'}) \in \mathcal{G}(\mathcal{A})$ implies there are traces h and h'

corresponding to $(\vec{\alpha},\vec{\beta})$ and $(\vec{\alpha'},\vec{\beta'})$. If $\vec{\alpha} \sqsubseteq \vec{\alpha'}$, then $\vec{\beta} \sqsubseteq \vec{\beta'}$ and the order of events in $\vec{\alpha}$ and $\vec{\beta}$ is the same in h and h' by monotonicity of deterministic Kahn networks.

(b) Iterated unbounded choice has P.
Let $\vec{\alpha} = (a_1 \cdots a_k)$ and $\vec{\alpha'} = (a_1 \cdots a_k \cdots)$ and $\vec{\beta'} = (b_1 \cdots b_k \cdots)$. (Iterated unbounded choice has a single input and a single output, so these are vectors of single sequences.) Then $\vec{\beta} = (b_1 \cdots b_k)$ is a possible output when the input is $\vec{\alpha}$ and there is a history of events in $\vec{\alpha}$ and $\vec{\beta}$ in the same order in $(\vec{\alpha},\vec{\beta})$ as in $(\vec{\alpha'},\vec{\beta'})$.

(c) Suppose 2 networks $N1$ and $N2$ have P. Construct a new network N by considering the input channels of N as all the input channels of $N1$ and $N2$. The output channels of N are all the output channels of $N1$ and $N2$. There are no connections between $N1$ and $N2$.

Let $\vec{\alpha}$ and $\vec{\gamma}$ be the inputs and let $\vec{\beta}$ and $\vec{\delta}$ be the outputs of $N1$ and $N2$ respectively. Let $(\vec{\alpha'};\vec{\gamma'},\vec{\beta'};\vec{\delta'}) \in \mathcal{G}(\mathcal{A})$ and $\vec{\alpha} \sqsubseteq \vec{\alpha'}$, $\vec{\gamma} \sqsubseteq \vec{\gamma'}$. Since $N1$ has P, $\exists\ (\vec{\alpha},\vec{\beta}) \in \mathcal{G}(A_{N1}) \cdot\ \vec{\beta} \sqsubseteq \vec{\beta'}$. In this, the relative order of events in $\vec{\alpha}$ and $\vec{\beta}$ is the same as the order of these events in $\vec{\alpha'}$ and $\vec{\beta'}$. Since $N2$ has P, $\exists\ (\vec{\gamma},\vec{\delta}) \in \mathcal{G}(A_{N2}) \cdot\ \vec{\delta} \sqsubseteq \vec{\delta'}$.

Shuffling the two corresponding histories in an appropriate way (i. e. maintaining order as before), we get a history of N and so $(\vec{\alpha};\vec{\gamma},\vec{\beta};\vec{\delta}) \in \mathcal{G}(A_N)$ and $\vec{\beta} \sqsubseteq \vec{\beta'}$, $\vec{\delta} \sqsubseteq \vec{\delta'}$.

(d) Suppose N has P. Let the input channels of N be $i_1 \cdots i_k$ and let the output channels of N be $o_1 \cdots o_l$. Connect o_1 (say) to i_1 (say) to get N'. Let $\vec{\alpha}$ and $\vec{\beta}$ be the inputs and outputs respectively to the modified network. Let $(\vec{\alpha'},\vec{\beta'}) \in \mathcal{G}(A_{N'})$ and let h be the corresponding trace. Let $\delta = \Pi_{o_1}(h)$. Then $(\vec{\alpha'};\delta,\vec{\beta'}) \in \mathcal{G}(A_N)$. Let $\vec{\alpha} \sqsubseteq \vec{\alpha'}$. Then $\vec{\alpha};\delta \sqsubseteq \vec{\alpha'};\delta$ and this implies $\exists(\vec{\alpha};\delta,\vec{\beta_1};\delta_1) \in \mathcal{G}(A_N)$ with $\vec{\beta_1} \sqsubseteq \vec{\beta}$ and $\delta_1 \sqsubseteq \delta$ (by the fact that N has P), and the order of events is the same as in the trace corresponding to $(\vec{\alpha'};\delta,\vec{\beta};\delta)$.

If $\delta = \delta_1$ then we are done (because then $(\vec{\alpha},\vec{\beta_1}) \in G(A_{N'})$, $\vec{\beta_1} \sqsubseteq \vec{\beta}$ and order preserved). Otherwise δ_1 is a proper prefix of δ. This implies that $\vec{\alpha};\delta_1$ is a prefix of $\vec{\alpha};\delta$, which implies $\exists(\vec{\alpha};\delta_1,\vec{\beta_2};\delta_2) \in \mathcal{G}(A_N)$ with $\vec{\beta_2} \sqsubseteq \vec{\beta_1} \sqsubseteq \vec{\beta}$ and $\delta_2 \sqsubseteq \delta_1$.

If we repeat the above step, we will reach $\delta_i = \delta_{i+1}$ and then we will be done. This procedure will terminate because the prefix ordering is a well-ordering and δ_i must certainly hit ϕ and stabilize if it does not stabilize earlier.

(e) Angelic merge does not have P.
Let the inputs to the two channels of angelic merge be 1^∞ and 2. Then the only possible outputs are 1^i21^∞ for $i \geq 0$, and 1^∞. Now extend the input to 1^∞ and 23 respectively. Now one possible output is 231^∞,but it is not an extension of 1^i21^∞ or 1^∞.

6 Conclusions

In this paper we have established that one cannot implement a fair merge operator in the context of static dataflow networks with angelic merge and with unbounded nondeterminism. This indicates that there is a separation of expressiveness power amongst various network primitives all of which feature unbounded non-determinism . It is clear from Koenig's Lemma arguments that one cannot produce unbounded indeterminacy from finite choice but there seemed to be no clear results indicating that unbounded indeterminacy was not enough to obtain fair merge. The question still remains open for dynamic networks but we feel that the methods used in the present work will suffice to establish that one cannot implement fair merge with *amb* in dynamic networks. Preliminary work indicates that this result holds . It is an easy exercise to implement both angelic merge and unbounded choice with *amb* in dynamic networks.

A related area of investigation into expressiveness is the finite delay operators of Milner's Calculus for Communicating Systems (CCS) [11]. One finds in the literature on CCS two different primitives for expressing finite delay, one due to Milner [12] and one due to Hennessy[4]. 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. Hennessy's finite delay operator resembles our poll operator in this respect. Preliminary work indicates that these are indeed inequivalent. This is interesting because CCS has aspects of dynamic networks. Similar suggestions

were made to us by Joachim Parrow [17].

The differences in expressiveness that we have uncovered seem to be part of a hierarchy that we have just begun to glimpse. In the course of our investigation we experimented with many variants of iterated unbounded choice and picked the version which was most easy to understand and motivate. There were, however, others which are not implementable by the primitives we have discussed in this paper. There appears to be a difference in expressive power between indeterminate primitives which "announce their choice" and those that do not. We are actively working on consolidating these results into a systematic hierarchy of indeterminacy primitives.

Acknowledgements

We have benefited from discussions with Samson Abramsky, Joachim Parrow, Elsa Gunter and especially Eugene Stark.

References

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

[2] J. Dean Brock and William B. Ackerman. Scenarios: a model of nondeterminate computation. In *Formalization of Programming Concepts*, pages 252–259, 1981. LNCS 107.

[3] Manfred Broy. Fixed point theory for communication and concurrency. In *Formal Description of Programming Concepts II*, pages 125–148, North-Holland, 1983.

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

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

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

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

[8] 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.

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

[10] Albert Meyer. Puzzles in the Logic of Programming. 1984. Available from Albert Meyer.

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

[12] Robin Milner. *A Finite-Delay Operator in Synchronous CCS.* Technical Report CSR-116-82, University of Edinburgh, 1982.

[13] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains i. *Theoretical Computer Science*, 13(1):85–108, 1981.

[14] Prakash Panangaden. *Denotational Semantics of Networks Containing Indeterminate Operators.* Master's thesis, University of Utah, 1985.

[15] Prakash Panangaden and Eugene Stark. *Computations, Residuals and the Power of Indeterminacy.* Technical Report 87-883, Dept of Computer Science, Cornell University, 1987.

[16] 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.

[17] Joachim Parrow. On delay operators in ccs. 1987. Private Communication.

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

[19] 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.

[20] Glynn Winskel. *Event Structures.* Technical Report 95, University of Cambridge, Computer Laboratory, 1986.