2 Process Networks

Contents

2.1 Kahn-MacQueen Process Networks **28**
2.2 Semantics of Process Networks **32**
Historical Notes: Process Networks 33
2.2.1 Least Fixed Point Semantics 34
2.2.2 Monotonic and Continuous Functions 38
2.3 Execution of Process Networks **43**
2.3.1 Turing Completeness of Process Networks 47
2.3.2 Effective Execution 47
2.4 Convergence of Execution to the Semantics **51**
2.5 Stable Functions **53**
2.6 Nondeterminism **54**
2.7 Rendezvous **54**
Sidebar: Limit of a Sequence of Real Numbers 54
2.8 Summary **55**
Exercises **56**

A **Kahn process network** (**KPN**) is a concurrent composition of sequential processes that communicate using a particular form of message passing. A key property of these networks is that, despite the concurrency, they are assured of being determinate, in the sense that a KPN defines a unique sequence of messages on each communication channel between processes. That is, the messages that are communicated do not depend on the scheduling of the sequential processes. This chapter develops the theory behind such

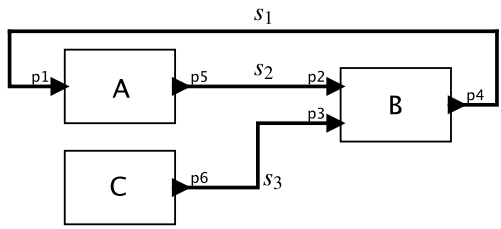


Figure 2.1: Example of a process network.

networks, explaining why they are determinate, and discussing constraints that an execution engine must satisfy to execute them correctly. For the historical background on this model of computation, see the sidebar on page 33. For a discussion of the practical use of process networks, see Ptolemaeus (2014).

2.1 Kahn-MacQueen Process Networks

A **process network** (**PN**) is a collection of components called **processes** or **actors**, each representing a sequential, step-by-step **procedure**. A, B, and C in Figure 3.1 depict processes as rectangles. A process may have **private state** (variables that are invisible to other processes). The steps in its procedure may manipulate its state, send messages to processes (including possibly itself), or receive messages from processes (again including possibly itself).

A process sends and receives messages via named input or output **ports**. In Figure 3.1, p1, p2, and p3 are input ports, and p4, p5, and p6 are output ports. An output port may be connected to an input port via a **channel** that carries a sequence of messages. Each message is called a **token**, and a sequence of tokens is called a **signal**. In Figure 3.1, s_1, s_2, and s_3 represent signals. A signal may consist of a finite or infinite sequence of tokens.

A key assumption in process networks is that tokens are delivered to the destination input port reliably and in order. The channel will **buffer** tokens for later delivery to the receiving process, and it will never lose tokens. A process can always send a token. It does not need to wait for the recipient to be ready to receive.

We begin with a special case of process networks that is easier to understand than the general case. This special case was introduced by Kahn and MacQueen (1977), and has been the subject of considerable study ever since. A **Kahn-MacQueen network** is a process network with **blocking reads**. This means that the only mechanism that a process has to receive a token is to attempt to read a token from an input port; that attempt blocks execution of the process if there is no available input token. The process will remain blocked until a token is available. In particular, a process cannot determine a priori whether a token is available. The only operation it has available is a blocking read.

A **Kahn-MacQueen process** is a program in an imperative language augmented with a **blocking read** statement and **nonblocking write** statement that reference the input and output ports of the process. Here, we describe Kahn-MacQueen processes using a C-like pseudo language that has structured control statements like `while` and `if-then-else`, plus statements like

```
t = read(p);
```

which performs a blocking read on input port p and returns a token `t`, and

```
write(p, t);
```

which writes the token `t` to port p. The `read` procedure call does not return until there is a token available at port p. The `write` statement returns immediately. Our language is a pseudo language in that we do not fully define it, and we omit important details like data types.

Example 2.1: Suppose that A in Figure 3.1 executes the following Kahn-MacQueen procedure,

```
write(p5, 0);
while(true) {
  t = read(p1);
  write(p5, t);
}
```

This procedure first produces an output token with value 0 and then enters an infinite loop where it reads an input token and sends it to the output port. If the input is a sequence tokens $(1,2,3)$, for example, then the output will be $(0,1,2,3)$.

An actor with this behavior is called a **unit delay** because the input tokens appear at the output delayed by one step in the sequence. In practice, the value of the initial output (0 in this case) would typically be a **parameter** of the actor rather than a built-in constant.

Let $a.b$ refer to the **concatenation** of sequences a and b. That is, $a.b$ is a sequence with **prefix** a followed by b. If a is infinite, then $a.b = a$. Using this notation, the process of Example 2.1 defines its output sequence as a function of its input sequence.

Example 2.2: If the input to the process of Example 2.1 is a sequence denoted by b, then the output is $A(b) = (0).b$.

We can use the definition of actor A from Example 2.1 to build a complete process network in Figure 3.1.

Example 2.3: Suppose that B executes the following procedure,

```
while(true) {
  t2 = read(p2);
  t3 = read(p3);
  write(p4, t2+t3);
}
```

Suppose further that C executes the following procedure,

```
while(true) {
  write(p6, 1);
}
```

An execution of the process network in Figure 3.1 yields the following sequences:

$$\begin{aligned} s_1 &= (1,2,3,4,\cdots) \\ s_2 &= (0,1,2,3,\cdots) \\ s_3 &= (1,1,1,1,\cdots) \end{aligned}$$

In the previous example, an execution of the process network yields infinite sequences. This is not always the case.

Example 2.4: Suppose that instead of the unit delay given in Example 2.1, A in Figure 3.1 executes the following Kahn-MacQueen procedure,

```
while(true) {
  t = read(p1);
  write(p5, t);
}
```

Such an actor is called an **identity actor**, because the output sequence is the same as the input sequence. An execution of the resulting process network yields

$$\begin{aligned} s_1 &= \bot \\ s_2 &= \bot \\ s_3 &= (1,1,1,1,\cdots) \end{aligned}$$

where $\bot$ is the empty sequence. Processes A and B both block immediately attempting to read a token provided by the other. Process C, on the other hand, is able to execute and produce an infinite sequence of outputs. Since the semantics of process networks requires that tokens on channels not be lost, the tokens produced by C must be stored until they are consumed by B. In this case, however, they will never be consumed by B, and eventually, any execution platform will run out of memory to store the tokens.

The previous example illustrates two phenomena that can occur with process networks. The first is **deadlock**, where the actors on a directed cycle of the network are blocked waiting for tokens for each other. In this case, we have a **local deadlock**, because only part of the process network is deadlocked. In particular, C is not blocked.

The second phenomenon illustrated by this example is **unbounded memory**. This process network cannot correctly execute without an unbounded amount of memory for storing unconsumed tokens.

We will see below that, in general, whether a Kahn-MacQueen process network deadlocks is undecidable, even if we constrain the actors to a few very simple primitive ones. It is

also undecidable whether such a process network can be executed with bounded memory. These two limitations are actually a consequence of the rich expressiveness of the model of computation. We will see that a very few primitive actors are sufficient to make the model of computation Turing complete, which means that it can describe every effectively computable function.

2.2 Semantics of Process Networks

Example 2.2 suggests that a Kahn-MacQueen process can be defined as a function that maps input sequences to output sequences. Consider an actor with a single input port and single output port. The **data type** of a port p is a set T_p of token values that the port consumes or produces. The set T_p^{**} is the set of finite and infinite sequences of tokens of type T_p (see Chapter 1). An actor is therefore a function defined on such sequences.

Example 2.5: Suppose that in Example 2.2, the input and output data types are $T = \mathbb{N}$, the natural numbers. Then the unit delay is a function of form

$$A\colon \mathbb{N}^{**} \to \mathbb{N}^{**}$$

where for all $b \in \mathbb{N}^{**}$,

$$A(b) = (0).b.$$

As we pointed out before, normally the initial output 0 would be a parameter of the actor rather than a built-in constant.

In general, the semantics of a program is its meaning. In the case of process networks, the semantics is the signals defined by the network. Each signal is a finite or infinite sequence of tokens. In this section, we will show that if processes define functions that satisfy a particular constraint (they are continuous), then the semantics of the network is unique (i.e., the network defines exactly one sequence of tokens for each signal), and we can give a constructive procedure for building the signals defined by the network (i.e., a mechanism for executing the network). For this section, you will want to have mastered the material on the prefix order and complete partial orders, in Chapter 1.

Historical Notes: Process Networks

Gilles Kahn (1946 – 2006), French computer scientist who developed Kahn process networks.

The notion of concurrent processes interacting by sending messages is rooted in Conway's **coroutines** (Conway, 1963). Conway described software modules that interacted with one another as if they were performing I/O operations. In Conway's words, "When coroutines A and B are connected so that A sends items to B, B runs for a while until it encounters a read command, which means it needs something from A. The control is then transferred to A until it wants to write, whereupon control is returned to B *at the point where it left off*." The key idea is that both A and B maintain as part of their state their progress through their procedures, and transfer of control occurs to satisfy demands for data.

The least fixed-point semantics is due to Kahn (1974), who developed the model of processes as continuous functions on a CPO. Kahn and MacQueen (1977) defined process interactions using non-blocking writes and blocking reads as a special case of continuous functions, and developed a programming language for defining interacting processes. Their language included recursive constructs, an optional functional notation, and dynamic instantiation of processes. They gave a demand-driven execution semantics, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976). Berry (1976) generalized these processes with stable functions.

Sequences of tokens communicated by process networks are **unbounded lists**. The notion of unbounded lists as data structures first appeared in Landin (1965). This underlies the communication mechanism between processes in a process network. The UNIX operating system, due originally to Ritchie and Thompson (1974), includes the notion of **pipes**, which implement a limited form of process networks.

Kahn (1974) stated but did not prove that a maximal and fair execution of process network yields the least fixed point. This was later proved by Faustini (1982) and Stark (1995). It has come to be known as the Kahn principle.

2.2.1 Least Fixed Point Semantics

Recall that a fixed point of a function $F : X \to X$ is an element $x \in X$ such that $F(x) = x$. Execution of every process network can be reduced to finding a fixed point of a function. The function is a composition of the functions defined by the individual actors.

Example 2.6: Consider the process network in Figure 3.1. This network is redrawn in Figure 2.2(a). In Figure 2.2(b), we reorganize the drawing and draw a box around the three actors. This box can itself be a considered an actor with three input ports and three output ports, as illustrated in Figure 2.2(c). Figure 2.2(d) further abstracts this by aggregating the three signals into one.

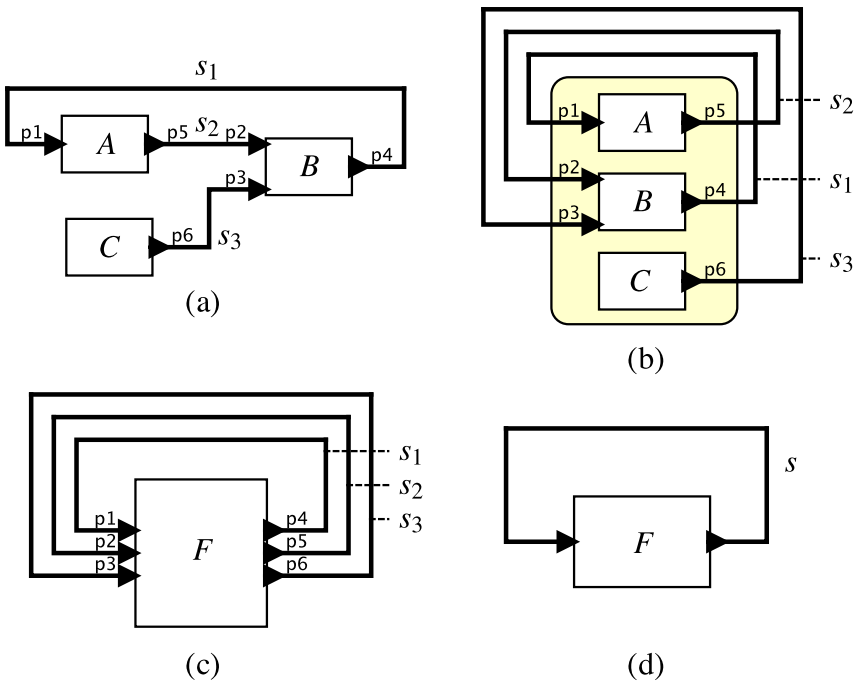


Figure 2.2: Execution of every process network can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

Suppose that the three signals in Figure 2.2 have data type T. Then each signal s_i, $i = 1,2,3$, is a member of the set T^{**} of sequences of tokens of type T. In Figure 2.2(d), the function F therefore has the form

$$F : (T^{**})^3 \to (T^{**})^3.$$

Using the process definitions from Examples 2.1 and 2.3, we see that if the input is

$$s = ((a_1, a_2, \cdots), (b_1, b_2, \cdots), (c_1, c_2, \cdots)),$$

a three-tuple of sequences, then the output is

$$F(s) = ((b_1 + c_1, b_2 + c_2, \cdots), (0, a_1, a_2, \cdots), (1, 1, \cdots)),$$

which is also a three-tuple of sequences.

Because of the feedback loop, we seek a sequence s that satisfies $F(s) = s$, so in this case, we require that

$$\begin{aligned} a_1 &= b_1 + c_1 \\ a_2 &= b_2 + c_2 \\ b_1 &= 0 \\ b_2 &= a_1 \\ c_1 &= 1 \\ c_2 &= 1 \\ \cdots & \end{aligned}$$

which implies that

$$\begin{aligned} a_1 &= 1 \\ a_2 &= 2 \\ b_1 &= 0 \\ b_2 &= 1 \\ c_1 &= 1 \\ c_2 &= 1 \\ \cdots & \end{aligned}$$

which is the same result we obtained before in Example 2.3.

Execution of a process network with n signals of type T can be reduced to finding a fixed point of a function of the form

$$F\colon (T^{**})^n \to (T^{**})^n.$$

If a process network has signals with different types, then the domain and codomain of the function will be a cartesian product of the sets of sequences of those types. The notation gets more complex, but not the concept. It may be surprising that *all* process networks can be reduced in the same way to finding a fixed point of a function.

Example 2.7: Consider the network in Figure 2.3(a). It may seem that this would be difficult to reduce to a fixed point problem because there is no feedback in the network. However, this can be redrawn as shown in Figure 2.3(b), and abstracted as shown in Figure 2.3(c). In this case, the function F is a rather trivial function. Regardless of the input signal, it always produces the same output signal. It is a **constant function**, where the output is independent of the input.

To be concrete, suppose that the data type is $T = \mathbb{N}$, the natural numbers, and that actor A produces the sequence $(0,1,2,3,\cdots)$. Then $F\colon \mathbb{N}^{**} \to \mathbb{N}^{**}$ is a function such that for all $s \in \mathbb{N}^{**}$,

$$F(s) = (0,1,2,3,\cdots).$$

Hence, $s = (0,1,2,3,\cdots)$ is a fixed point of F (the only fixed point, in fact).

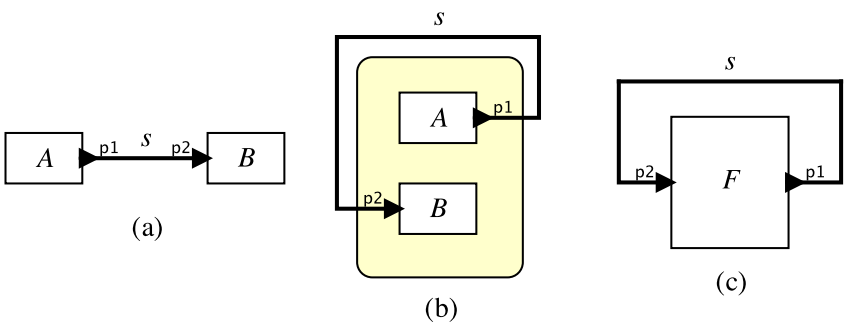


Figure 2.3: Even a process network with no feedback loops can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

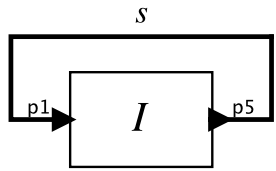


Figure 2.4: A process network with multiple fixed points, where I is the identity actor.

For the above examples, the function F has exactly one fixed point. This is not always the case.

Example 2.8: Consider an identity actor I, described in Example 2.4, where for all $s \in T^{**}$,

$$I(s) = s.$$

Suppose that this actor is used the process network shown in Figure 2.4. This process network has multiple fixed points. In fact, any signal $s \in T^{**}$ is a fixed point.

Recall from Chapter 1 that the prefix order is a partial-order relation $\sqsubseteq$ over sets of sequences T^{**}. For any two $a, b \in T^{**}$, $a \sqsubseteq b$ if the sequence a forms the leading part of sequence b. If a is infinite, then $a \sqsubseteq b$ if and only if $a = b$.

When a process network has multiple fixed points, we will choose the **least fixed point**, where "least" is in the prefix order.

Example 2.9: For the process network in Figure 2.4, the least fixed point is $s = \perp$, the empty sequence. This choice is indeed the least fixed point because for all other fixed points s, $\perp \sqsubseteq s$. Moreover, it is exactly the fixed point that results from executing the Kahn-MacQueen process! The process deadlocks immediately, blocked on the first read. Hence, the result of execution is the empty sequence.

We will see next that with Kahn-MacQueen process networks, the least fixed point is always uniquely defined, and execution of the network produces that least fixed point. We will then show that some actors that cannot be defined as Kahn-MacQueen processes also yield a uniquely defined least fixed point.

2.2.2 Monotonic and Continuous Functions

Recall that a function $A\colon T^{**} \to T^{**}$ is monotonic if $a \sqsubseteq b$ implies that $A(a) \sqsubseteq A(b)$. A Kahn-MacQueen process defines a function from input sequences to output sequences as long as the imperative language used to define the language is determinate. This means that given particular input signals at the input ports, the output signals are fully defined by the process. More interestingly, Kahn-MacQueen processes are monotonic.

Example 2.10: The unit delay function considered in Example 2.5 is monotonic. Consider two possible input signals $a, b \in \mathbb{N}^{**}$. The corresponding output sequences are $A(a) = (0).a$ and $A(b) = (0).b$. Clearly, if $a \sqsubseteq b$, then $A(a) \sqsubseteq A(b)$.

The fact that Kahn-MacQueen processes are monotonic is easy to see by considering their execution. Suppose $a, b \in \mathbb{N}^{**}$ are two possible input signals where $a \sqsubseteq b$. A Kahn-MacQueen process that is presented with input b will first read a sequence of tokens equal to a. Since the process its determinate, once it has read this prefix of b, its output will be exactly the output sequence it would have produced had it been presented with input a. Upon continuing to read tokens from b (which now differ from a), it will only extend the output signal. It has no mechanism for retracting previously produced tokens, and hence the output produced in response to b is an extension of the output produced in response to a.

Notice that the blocking reads of a Kahn-MacQueen process play an important role in ensuring monotonicity.

Example 2.11: Consider an actor A with one input port and one output port that produces on its output the sequence (0) if the input is the empty sequence, and otherwise produces the output (1). This actor clearly defines a function, in that

the output is fully defined by the input. However, this function is not monotonic. Suppose that $a = \bot$, the empty sequence, and $b = (0)$. Then $a \sqsubseteq b$, but $A(a) = (0)$ is not a prefix of $A(b) = (1)$.

Notice that this actor cannot be implemented as a Kahn-MacQueen process. In order to produce the output (0), the actor needs to know that that input is the empty sequence. But the only operation it has available on the input port is to perform a blocking read. If the input is indeed empty, then it will remain blocked forever, and hence will not be able to produce the output (0).

As explained in Example 1.15 of Chapter 1, the poset $(T^{**}, \sqsubseteq)$ is a complete partial order (CPO). Moreover, by Proposition 1.3, the poset $((T^{**})^n, \sqsubseteq)$ is also a CPO, where $\sqsubseteq$ is the pointwise prefix order. A function $F\colon (T^{**})^n \to (T^{**})^n$ is continuous if for all chains $C \subseteq (T^{**})^n$,

$$F(\bigvee C) = \bigvee \hat{F}(C) \ ,$$

where $\hat{F}$ is the lifted version of F. By Proposition 1.5, every continuous function is monotonic.

In practice, every Kahn-MacQueen process is not only monotonic, but also continuous. Why? Intuitively, continuity means that a function does not "wait forever" before producing output. Suppose that the chain $C = \{s_0, s_1, s_2, \cdots\}$ represents a sequence of partially constructed inputs to a Kahn-MacQueen process. Then $\bigvee C = s$ represents the eventual, complete input. The set

$$\hat{F}(C) = \{F(s_0), F(s_1), F(s_2), \cdots\}$$

represents the partially constructed outputs, given partially constructed inputs. Then continuity requires that $\bigvee \hat{F}(C)$ be equal to the eventually complete output $F(s)$.

Example 2.12: Suppose that A is an actor with one input port and one output port. Suppose that its input is eventually going to be the infinite sequence

$$s = (0, 1, 2, 3, \cdots)$$

Suppose that $C = \{s_0, s_1, s_2, \cdots\}$, where

$$\begin{aligned} s_0 &= (0) \\ s_1 &= (0,1) \\ s_2 &= (0,1,2) \\ \cdots \end{aligned}$$

C is clearly a chain. Moreover,

$$\bigvee C = s.$$

Suppose further that A is the function

$$A(s) = \begin{cases} \bot & \text{if } s \text{ is finite} \\ (1) & \text{otherwise} \end{cases}$$

This function is clearly monotonic, but it is not continuous. In particular, when the function is applied to each partially constructed input in C, the output is $\bot$. Hence,

$$\hat{F}(C) = \{F(s_0), F(s_1), F(s_2), \cdots\} = \{\bot\}.$$

Hence,

$$\bigvee \hat{F}(C) = \bot\,.$$

However,

$$F(\bigvee C) = F(s) = (1).$$

These two are not equal. Intuitively, the function F has to wait forever to determine whether the input is finite or not.

Notice that this function cannot be implemented by a Kahn-MacQueen process. We might try to specify it as follows,

```
while(true) {
   read(p1);
}
write(p2, 1);
```

That is, after reading an infinite number of inputs, we output a 1. If the input is finite, this process will block forever on one of the reads. But in practice, it will never reach the point of producing the 1, so this procedure does not really implement the function.

The Kleene fixed-point theorem (Proposition 1.7) then provides what we need. It states that if F is a continuous function, then it has a unique least fixed point. Moreover, it asserts that the least fixed point is the result of applying the function first to $\perp$, and then recursively to the result, etc., as follows:

$$\begin{aligned} s_0 &= \perp \\ s_1 &= F(\perp) \\ s_2 &= F(F(\perp)) \\ \cdots & \\ s_m &= F^m(\perp) \\ \cdots & \end{aligned}$$

Thus, instead of solving a system of equations, as we did in Example 2.6, we can construct the solution by just repeatedly applying the function F.

Example 2.13: Consider the same process network of Example 2.6. Recall that the function $F\colon (T^{**})^3 \to (T^{**})^3$ is given by

$$F(s) = ((b_1 + c_1, b_2 + c_2, \cdots), (0, a_1, a_2, \cdots), (1, 1, \cdots)),$$

where the input is

$$s = ((a_1, a_2, \cdots), (b_1, b_2, \cdots), (c_1, c_2, \cdots)),$$

This function is easily shown to be continuous.

In this case, the bottom of the CPO is the three-tuple of empty sequences, so the constructive procedure given by the Kleene fixed-point theorem proceeds as follows,

$$\begin{aligned} F((\perp,\perp,\perp)) &= (\perp, (0), (1,1,\cdots)) \\ F(F((\perp,\perp,\perp))) &= ((1), (0,1), (1,1,\cdots)) \\ F(F(F((\perp,\perp,\perp)))) &= ((1,2), (0,1,2), (1,1,\cdots)) \\ & \cdots \end{aligned}$$

This eventually converges to the same sequences found in Example 2.3.

Notice that this procedure immediately creates a practical problem. The very first invocation of the function F yields as the third element of the output tuple an infinite sequence. If we were to literally implement this procedure in a computer, then we would never get past the first step, and we would exhaust available memory in any attempt to generate this infinite sequence. The following section will consider practical execution policies.

Example 2.14: For the variant given by Example 2.4, which has a local deadlock, the Kleene procedure immediately converges to the final answer,

$$F((\bot,\bot,\bot)) = (\bot,\bot,(1,1,\cdots)).$$

Example 2.15: For the process network in Figure 2.4, recall that the least fixed point is $s = \bot$. The Kleene procedure immediately converges to this solution.

Example 2.16: Recall that the unit delay actor of Example 2.1 is a function of form

$$A\colon \mathbb{N}^{**} \to \mathbb{N}^{**}$$

where for all $b \in \mathbb{N}^{**}$,

$$A(b) = (0).b.$$

This function is continuous. Suppose we put it in a feedback loop, as we did with the identify function in Figure 2.4. Then the Kleene procedure would yield

$$\begin{aligned} A(\bot) &= (0) \\ A(A(\bot)) &= (0,0) \\ A(A(A(\bot))) &= (0,0,0) \\ &\cdots \end{aligned}$$

The least upper bound of this chain is the infinite sequence $(0,0,0,\cdots)$.

2.3 Execution of Process Networks

The least fixed point of a process network often includes infinite sequences. In practice, no computer program can construct an infinite sequence. Our interpretation is that process networks whose semantics include infinite sequences do not terminate. They are able to continue to execute for as long as we would like them to execute. But at any point during the execution, the sequences that they have produced are mere approximations of the semantics of the network. When there are multiple infinite sequences defined by a network, then there are many possible approximations. In this section, we consider execution policies that provide useful approximations.

First, we define a **correct execution** of a process network to be one which, at any time during execution, has constructed a prefix of each signal defined by the semantics of the network. These prefixes will always be finite, even if the semantics of the network includes infinite sequences.

Second, we define a **maximal execution** to be a correct execution that either does not halt, or if it halts, has produced exactly every sequence defined by the network. What we mean by "halt" in this case is that the execution no longer appends tokens to signals. A maximal execution will only halt if the semantics of the network defines only finite signals. An **infinite execution** is an execution that does not halt. All infinite executions are maximal.

Third, we define a **fair execution** to be one that ensures that if any process is able to produce an output token or read an input token, then it will eventually be allowed to do so.

The following proposition, known as the **Kahn principle**, stated by Kahn and MacQueen (1977) and later proved by Faustini (1982) and Stark (1995), provides key guidance for execution policies.

Proposition 2.1. *Any two fair and maximal executions of a process network produce the same sequences of tokens, matching the least fixed point in the Kahn semantics.*

The Kahn principle seems to suggest that practical executions of process networks should always be fair and maximal. This is not in fact obvious. The following example shows that fair and maximal executions are not always desirable.

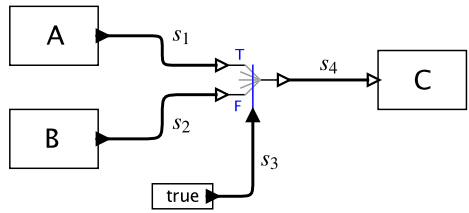


Figure 2.5: Example of a process network where unfair execution is desirable because fair execution will exhaust available memory.

Example 2.17: Consider the process network in Figure 2.5. Suppose that actors A and B produce infinite sequences, and C is able to consume an infinite sequence. Suppose that the actor labeled true produces an output of type $\mathbb{T} = \{true, false\}$ that is a constant infinite sequence

$$s_3 = \{true, true, true, \cdots\}.$$

The remaining actor in the center, which has no label, is a **Select** actor. It accepts a sequence of control tokens on the bottom port, and uses these to merge the sequences at its other two input ports. The merged sequence is the output. Specifically, the Select actor has three input ports, T, F, and control, where the control port is shown on the bottom of its icon. It has a single output port p. The actor is defined by the following Kahn-MacQueen procedure,

```
while{true} {
  c = read(control);
  if (c) {
    t = read(T);
  } else {
    t = read(F);
  }
  write(p, t);
}
```

Any fair execution of this process network has to permit B to produce its infinite sequence of tokens, even though those tokens will never be consumed. This will require unbounded memory to store unconsumed tokens.

There is, however, a maximal execution of this network that runs in bounded memory. Such an execution would only permit actor B to produce a finite number of tokens. Such an execution is maximal and correct, but not fair.

A **bounded execution** is an execution of a process network where there is a natural number $M \in \mathbb{N}$ such that at all times during the execution there are no more than M unconsumed tokens. A fair and maximal execution of the network in Figure 2.5 is not bounded.

An **effective process network** is one where every token that can be produced by an actor will eventually be read by the destination actor(s) (Geilen and Basten, 2003). The process network in the previous example is not effective. For networks that are not effective, we do not necessarily want fair and maximal execution, because such execution will eventually fail due to running out of memory.

Even for networks that are effective, executing them with bounded memory is not always easy. Suppose that we modify the network in Figure 2.5 so that the actor labeled true produces a random sequence of *true* and *false* values. How can we ensure that A and B so not overflow available memory on their outputs? Even for much simpler networks there is risk of memory overflow.

Example 2.18: Suppose that in the network in Figure 2.3(a), actor A produces tokens faster than actor B. Then eventually, we will run out of memory.

We seek, therefore, an execution policy that will deliver maximal executions for all process networks, bounded executions for networks for which bounded execution is possible, and fair executions for effective process networks. We call these **effective executions**. Effective executions are not required to be fair for networks that are not effective. They are not required to be bounded for networks that have no bounded execution. They are required to be maximal for all networks.

There is a long history of failed attempts to achieve this goal. One popular proposed technique is **demand-driven** execution, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976), where no token is produced unless there is a downstream actor that is ready to consume it. Unfortunately, this does not solve the problem, as illustrated by the following example.

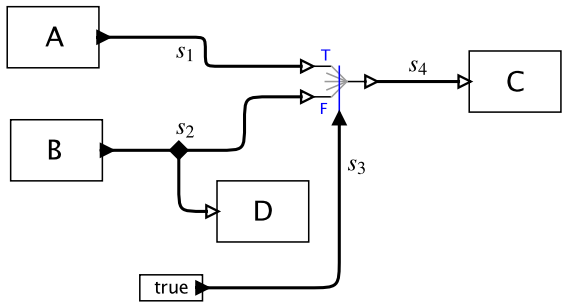


Figure 2.6: Example of a process network where demand-driven execution fails to deliver a bounded execution.

Example 2.19: A variant of Figure 2.5 is shown in Figure 2.6. The only difference is that is a new actor D has been added that consumes tokens from actor B. Each token produced by B needs to be routed to both the Select actor and to D. This can be accomplished by a **fork** process, indicated in the figure by a small black diamond. The fork can be implemented, for example, by the Kahn-MacQueen procedure

```
while (true) {
    t = read(in);
    write(out1, t);
    write(out2, t);
}
```

Thus, there will be two buffers, one storing tokens destined for the Select, and one storing tokens destined for D.

Assume that D is able to read an infinite input sequence. As a consequence, it will demand tokens. In demand-driven execution, the fork will respond by demanding inputs. Once the fork receives an input, the above Kahn-MacQueen implementation will send the token to both outputs, and hence the buffer at the F input of the Select will eventually overflow. If the fork is implemented differently so that it only sends outputs in response to demands, then it will be obligated to store the token locally in case the token is later demanded. This too will overflow available memory.

In the previous example, demand-driven execution can deliver bounded execution only if we somehow regulate the demands issued by the two sink actors, C and D. But how would we do that? Moreover, suppose that network contains cycles, or more interestingly, a multiplicity of cycles. How would demands be generated?

Data-driven execution is the complement of demand-driven execution. Instead of **sink** actors driving the execution (actors with no output ports), **source** actors drive the execution (actors with no input ports). An actor is permitted to execute only when it has available input data, except source actors, which are always permitted to execute. This obviously does not achieve the goal, since even a network as simple as that in Figure 2.3(a) cannot be assured of remaining bounded.

The problem of achieving effective executions is harder than it first appears. Fair execution, demand-driven execution, and data-driven execution all fail. It is not surprising that the problem is hard, however. It turns out that whether a bounded execution exists is undecidable, as is whether an infinite execution exists. We defend this claim in the next subsection. After that (Section 2.3.2), we show how to solve the undecidable problem by giving a policy that delivers effective execution.

2.3.1 Turing Completeness of Process Networks

(FIXME: Fill in this section)

2.3.2 Effective Execution

A partial solution to the problem of delivering effective execution was given by Parks (1995). Parks' solution delivers bounded and maximal execution for every process network that has a bounded execution. But it does not guarantee a fair execution.

Parks' algorithm is simple. An execution starts with bounded buffers on all connections between actors. It does not matter what the bound is. When a process attempts to write to a buffer that is full, the write blocks. That is, the process stalls, waiting for room to become available. Now, one of three things can happen. First, the network might execute forever, continuing to produce and consume tokens, even though some processes are blocked on a write. This achieves an infinite (hence maximal) bounded execution. Second, the network might deadlock where all processes have either terminated or are blocked on a read. In this case, the process network specified a finite execution, and the

finite execution has been completed. The execution is again maximal and bounded. Third, the network might deadlock where at least one process is blocked on a write. In this third case, Parks' strategy finds the smallest buffer on which a process is write blocked and increases its capacity so as to unblock the write-blocked process. Execution continues until again one these three scenarios occurs.

For many process networks, Parks' strategy achieves an effective execution. But not for all, as illustrated by the following example.

Example 2.20: Consider the process network in Figure 2.7. Suppose that all four processes can produce and consume infinite signals. Moreover, suppose that C is defined by the following Kahn-MacQueen procedure,

```
while (true) {
  t = read(in);
  write(out1, t);
  write(out1, t);
  write(out2, t);
}
```

Suppose further that D is defined by the following Kahn-MacQueen procedure,

```
write{out, 0);
while (true) {
```

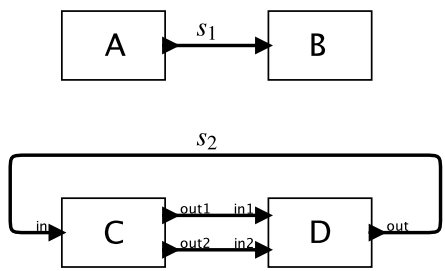


Figure 2.7: Example of a process network where Parks' algorithm does not yield an effective execution.

```
3     t = read(in2);
4     t = read(in1);
5     t = read(in1);
6     write(out, t);
7   }
```

It is easy to see that the least fixed point yields the infinite signal $s_2 = (0,0,0,\cdots)$.

Now suppose that we execute this network using Parks' algorithm, starting with initial buffer sizes of 1. D successfully produces its first output, enabling C to complete the read on line 2. C will then perform the write on line 3, but the write on line 4 will block because the buffer capacity has been reached. As a consequence, D will block on the read on line 3. A local deadlock occurs.

Under Parks' algorithm, no corrective action is taken because actors A and B are able to continue running indefinitely. As a consequence, this is not a fair execution. To be fair, every actor that is able to produce outputs must be allowed to do so. C is able to produce outputs, but we have blocked it artificially as part of our execution policy (for this reason, the state of the execution is called a **local artificial deadlock**).

Geilen and Basten (2003) identified this flaw in Parks' algorithm, and proposed an alternative execution strategy. Their strategy is similar to that of Parks in that execution starts with arbitrarily bounded buffers. But while Parks' algorithm looks only for a **global artificial deadlock** (all processes are blocked, at least one on a write), Geilen and Basten's strategy looks for local artificial deadlock.

Specifically, if at any time during the execution there is a sequence of actors $(A_1, A_2, \cdots, A_n)$, $n \geq 1$, where A_i is connected to A_{i+1} and A_n is connected to A_0, where all of these actors are blocked, at least one on a write, then we have a local artificial deadlock. Note that when we say "A_i is connected to A_{i+1}," we mean that either A_i send tokens to A_{i+1} or vice versa. So $(A_1, A_2, \cdots, A_n)$ is an (undirected) cycle of blocked actors. When an artificial deadlock is detected, the Geilen and Basten strategy will increase the size of at least one buffer on which a process in the cycle is write blocked.

Geilen and Basten (2003) prove that this strategy delivers an effective execution for effective process networks. But they make no guarantee about networks that are not effective. In fact, for the network in Figure 2.5, the strategy will overflow memory. Specifically, ac-

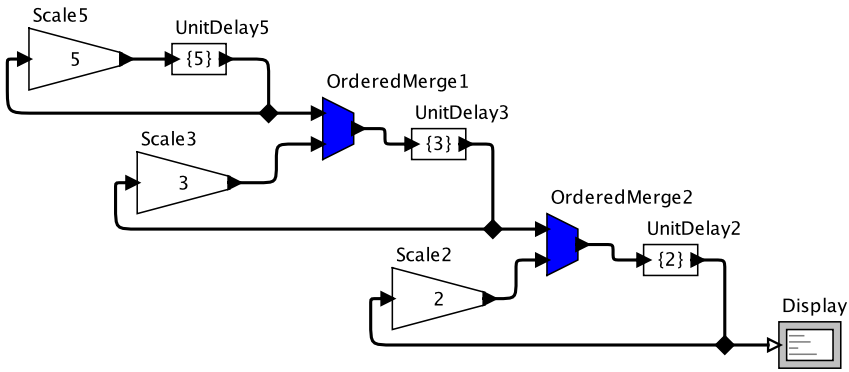


Figure 2.8: A process network for which it is difficult to determine appropriate buffer sizes.

tor B alone forms an undirected cycle (trivially), so whenever it write blocks, the capacity of its output buffer will be increased. Since the tokens it writes to the buffer are never read, the buffer will grow without bound.

For all of the examples we have considered so far, it is relatively easy to determine whether a bounded execution exists. This is not always the case, as illustrated by the following example.

Example 2.21: Consider for example the network shown in Figure 2.8. The output is an ordered sequence of integers of the form $2^n 3^m 5^k$, where n, m and k are non-negative integers. These are known as the **Hamming numbers**, and this program for computing them was studied by Dijkstra (1976) and Kahn and MacQueen (1977).

To understand how this network works, we need to understand what each of the actors does. The ones labeled ScaleN simply read an integer-valued input token, multiply it by N, and send the product to the output port. They repeat this sequence forever, as long as there are input tokens. The UnitDelay actors are parameterized versions of the unit delay that we saw in Example 2.1. They first produce an output token with value N, then repeat forever the following sequence: read the input token and send it to the output. As before, the black diamonds are fork actors. They direct a single token sequence to multiple destinations (over separate, unbounded buffers).

The processes labeled OrderedMergeM are the only nontrivial processes in this program. Given a sequence of numerically increasing tokens on each input port, they produce a numerically increasing merge of the two input sequences on their output ports (see Exercise 4 for a Kahn-MacQueen implementation of such an ordered merge). You can now understand how this program generates the Hamming numbers. The loop at the upper left produces the sequence

$$(5, 25, 125, \cdots, 5^n, \cdots).$$

That sequence is merged into the the second loop, which as a result produces the sequence

$$(3, 5, 9, 15, 25, \cdots, 5^n 3^m, \cdots.$$

That sequence is merged into the final loop, which produces the Hamming numbers, in numerically increasing order, without repetitions.

Although it is far from easy to see at a glance, it is possible to prove that any finite bound on buffer sizes will cause the network to fail to produce all the Hamming numbers. This network intrinsically requires unbounded buffers. It is effective, and both Parks' and Geilen and Basten's strategies deliver an effective execution.

2.4 Convergence of Execution to the Semantics

The Kahn principle states that maximal and fair executions produce sequences that match that the least fixed point. But what do we mean by "match?" Every execution is finite at all times, and can only have produced a prefix of the least fixed point. Intuitively, we would like such executions to converge in the limit to the least fixed point. But to make this intuition precise, we have to define a notion of convergence for sequences.

We use the notion of a topology. Let X be any set. A collection τ of subsets of X is called a **topology** if three conditions are satisfied:

1. X and $\emptyset$ are members of τ.
2. The intersection of any two members of τ is in τ.
3. The union of any collection of members of τ is in τ.

For any topology τ, the members of τ are called the **open set**s of the topology. See the sidebar on page 54 for an explanation of how this relates to the usual notion of open sets of real numbers. In our case, we are not concerned with real numbers, but rather with sequences of tokens of arbitrary type.

There is an important but subtle distinction between conditions (2) and (3) above. A "collection of members of τ" may be infinite. Thus, in a topology, the *union* of an infinite number of open sets is an open set. But the *intersection* of an an infinite number of open sets may not be an open set. The intersection of any *finite* number of open sets is required to be an open set, but not of an *infinite* number of open sets.

Consider a set T and the set T^{**} of finite and infinite sequences of elements of T. Given a *finite* sequence $s \in T^{**}$, an **open neighborhood** N_s around s is the set

$$N_s = \{s' \in T^{**} \mid s \sqsubseteq s'\}.$$

N_s is the set of sequences with prefix s.

Let τ be the collection of all sets that are arbitrary unions of such open neighborhoods. Then τ is a topology. This topology is a particular kind of **Scott topology**, named after Dana Scott (1932–present), a renowned computer scientist and mathematician. See Exercise 9 for some properties of this topology.

Notice that an open neighborhood is defined by a *finite* prefix. Were we to allow an infinite prefix, then a singleton set consisting of a single infinite sequence would be an open neighborhood. So infinite prefixes are not allowed.

Note further that an arbitrary intersection of open sets is not necessarily an open set. Consider any infinite chain $C \subset T^{**}$ containing only finite sequences. Each $c_i \in C$ defines an open neighborhood (the set of all sequences with c_i as a prefix). The intersection of all of these open neighborhoods can only contain sequences that have every $c_i \in C$ as a prefix. There is only one such sequence, $\bigvee C$, so this intersection of open sets is a singleton set containing exactly one infinite sequence. This singleton set is not an open set.

Using our Scott topology, we can now define a notion of convergence of a sequence of sequences. Consider a sequence of sequences

$$S\colon \mathbb{N} \to T^{**}.$$

This sequence is said to converge to a sequence $a \in T^{**}$ if for all open sets A containing a, there exists an integer $n \in \mathbb{N}$ such that for all integers $m > n$, the following holds:

$$S(m) \in A.$$

Intuitively, this means that for any finite prefix $p \sqsubseteq a$, the sequences in S eventually all have prefix p.

In summary, in the Scott topology, for a sequence of sequences to converge to an infinite sequence, it is necessary for the sequences to grow without bound. They need to eventually produce every finite prefix of the infinite sequence.

This notion of convergence creates an interesting conundrum. A **convergent execution** of a process network is one that eventually produces every finite prefix of the least fixed point. A maximal execution, by contrast, is only required to produce some prefix of the least fixed point. A maximal execution does not necessarily converge to the least fixed-point.

Example 2.22: For the example in Figure 2.5, a maximal execution is permitted to avoid executing actor B. In particular, the output s_3 of actor true is an infinite sequence of boolean `true` tokens, so the output s_2 of actor B is never needed and will never be consumed by the Select. But if B is capable of producing an infinite sequence, then the least fixed point solution for s_2 is an infinite sequence. Any execution that converges to this least fixed point solution will exhaust available memory because tokens produced by B will not be consumed.

Maximal and fair executions, however, do converge. For effective process networks, Geilen and Basten's execution strategy, described in Section 2.3.2, will deliver an effective execution, and that effective execution will converge to the least fixed point. For networks that are not effective, however, the strategy is less satisfactory because it may exhaust available memory. For such networks, Parks' algorithm may be preferable, even though it does not converge to the least fixed point.

2.5 Stable Functions

(FIXME: Section on functions that are not sequential)

2.6 Nondeterminism

(FIXME: nondeterminate merge and the Brock-Ackerman anomaly.)

2.7 Rendezvous

(FIXME: Rendezvous as PN with buffer size zero.)

Sidebar: Limit of a Sequence of Real Numbers

An **open neighborhood** around a in $\mathbb{R}$ is the set

$$N_{a,\varepsilon} = \{x \in \mathbb{R} \mid a - \varepsilon < x < a + \varepsilon\}$$

for some positive real number ε. An **open set** A in the reals is a subset of $\mathbb{R}$ such that for all $a \in A$, there is an open neighborhood around a that is a subset of A. The collection of such open sets in the reals is called the **standard topology** for the reals. It is a topology because it includes both $\mathbb{R}$ itself and $\emptyset$, it includes the intersection of any two such open sets, and it includes an arbitrary union of such open sets. Notice that it *does not* include any arbitrary intersection of open sets. Consider for example the intersection of the sets

$$N_{0,\varepsilon}, \quad \text{where } \varepsilon = 1/n, \quad n \in \{1, 2, 3, \cdots\}.$$

The intersection of these sets is the singleton set $\{0\}$, which is not an open set.

Consider a sequence of real numbers

$$s \colon \mathbb{N} \to \mathbb{R}.$$

This sequence is said to converge to a real number a if for all open sets A containing a there exists an integer $n \in \mathbb{N}$ such that for all integers $m > n$ the following holds:

$$s(m) \in A.$$

This notion of convergence captures the intuition that the sequence eventually gets arbitrarily close to a.

2.8 Summary

(FIXME: Summary here)

Exercises

1. Given a set A and the CPO $(A^{**}, \sqsubseteq)$, for each of the following functions, state whether it is monotonic, continuous, both, or neither. Assume the domain and codomain of every function is A^{**}. Assume the period operator represents concatenation of sequences.

   (a) The **unit delay** function d given by $\forall\, s \in A^{**}$, $d(s) = (a).s$ where $a \in A$.

   (b) The **trailer function** t given by, $\forall\, s \in A^{**}$,

   $$t(s) = \begin{cases} s.a & \text{if } s \text{ is finite} \\ s & \text{otherwise} \end{cases}$$

   where $a \in A$.

   (c) The **is finite function** f given by, $\forall\, s \in A^{**}$,

   $$f(s) = \begin{cases} (a) & \text{if } s \text{ is finite} \\ (a') & \text{otherwise} \end{cases}$$

   where $a, a' \in A$, $a \neq a'$.

   (d) An alternative **is finite function** f' given by, $\forall\, s \in A^{**}$,

   $$f(s) = \begin{cases} (a) & \text{if } s \text{ is finite} \\ (a,a) & \text{otherwise} \end{cases}$$

   where $a \in A$.

   (e) Let $m\colon (A^{**})^2 \to A^{**}$ be the **fair alternating merge** function, defined as follows. Given two infinite sequences $s_1 = (a_0, a_1, \cdots)$ and $s_2 = (b_0, b_1, \cdots)$ it outputs the infinite sequence $m(s_1, s_2) = (a_0, b_0, a_1, b_1, \cdots)$. That is, it alternates the elements of the sequences. If either or both of the inputs is finite, then it alternates their elements until the shorter of the two runs out of elements, and then it outputs the remaining values from the longer of the two. For example, if $s_1 = (a_0, a_1, \cdots, a_n)$ is finite, but s_2 is infinite, then it produces

   $$m(s_1, s_2) = (a_0, b_0, a_1, b_1, \cdots, a_n, b_n, b_{n+1}, b_{n+2}, \cdots)\ .$$

   If s_2 is also finite, but longer than s_1, then the result will be similar to the above, but finite, ending with the last value of s_2

(f) Let $m\colon (A^{**})^2 \to A^{**}$ be the **unfair merge** function, defined as follows.

$$m(s_1, s_2) = \begin{cases} s_1 & \text{if } s_1 \text{ is infinite} \\ s_1.s_2 & \text{otherwise} \end{cases}$$

2. Consider a function $f\colon \mathbb{B}^{**} \to \mathbb{B}^{**}$, where $\mathbb{B} = \{0,1\}$ is the set of binary digits. Given an input sequence, this function outputs the same sequence, but with a 0 inserted at the start and between any two tokens. That is, if the input is

$$s = (b_1, b_2, \cdots, b_n),$$

then the output will be

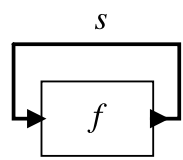
$$f(s) = (0, b_1, 0, b_2, \cdots, 0, b_n).$$

If the input is infinite, then the pattern simply continues. Assume that $f(\bot) = (0)$. Consider the poset $(B^{**}, \sqsubseteq)$, where $\sqsubseteq$ is the prefix order.

(a) Is f monotonic in $(B^{**}, \sqsubseteq)$? Continuous?

(b) Give a Kahn-MacQueen process implementation of the function f, or argue that there is no such implementation.

(c) Consider a use of this function in the following process network:



Give the least fixed point solution s, or show that there is no least fixed point solution s.

(d) Can the feedback network above execute forever with bounded memory? Explain.

Repeat (a) through (d) where instead of assuming that $f(\bot) = (0)$, you assume that $f(\bot) = \bot$.

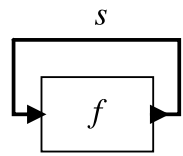
3. Consider a function $f\colon \mathbb{B}^{**} \to \mathbb{B}^{**}$ given by

$$f(s) = \begin{cases} (0), & \text{if } s = \bot \text{ or } (0) \sqsubseteq s \\ (1), & \text{if } (1) \sqsubseteq s \end{cases}$$

for all $s \in \mathbb{B}^{**}$, where $\mathbb{B} = \{0,1\}$ is the set of binary digits.

(a) Is f monotonic? Continuous? Justify your answer.

(b) Consider a use of this function in the following process network:



Give the least fixed point solution s, or show that there is no least fixed point solution s.

(c) Give a Kahn-MacQueen process implementation of the function f, or argue that there is no such implementation.

4. Exercise 2.21 considers the process network in Figure 2.8, which includes an OrderedMerge actor. Give a Kahn-MacQueen procedure implementing the OrderedMerge. Assume two inputs named in1 and in2 and one output named out.

5. The LUB of a chain can be interpreted as a limit, but it is not always the limit that one would expect intuitively. This problem explores this issue for the set $T = \{t, f\}$ and the CPO $(A = T^{**}, \sqsubseteq)$, and offers an alternative way to construct limits.

(a) As a warm up, find the LUB of the following set, or show that it has no LUB:

$$\{(t), (t,f), (t,f,t), (t,f,t,f), \ldots\}.$$

(b) Show that the following set has no LUB:

$$\{a_1, a_2, a_3, \ldots\} = \{(t,f), (t,t,f), (t,t,t,f), \ldots\}.$$

(c) Given a set A, a metric is a function $d\colon A \times A \to \mathbb{R}$ such that three properties are satisfied for all $a, b, c \in A$:

i. $d(a,b) = d(b,a)$,

ii. $d(a,b) = 0 \iff a = b$ and

iii. $d(a,b) + d(b,c) \geq d(a,c)$.

With $A = T^{**}$, we can define a metric d where for all $a, b \in A$,

$$d(a,b) = 1/n$$

where n is the index of the first position at which a and b differ, and is zero if $a = b$. For example, $d((t),(f)) = 1$ and $d((t),(t,f)) = 1/2$. Show that d is a metric.

(d) For the set in part (b), a_i is defined to be the sequence starting with i instances of t followed by one f. Let

$$a = (t, t, t, \ldots)$$

(an infinite sequence of t). Show that a_i converges to a as $i \to \infty$ in the sense that for any real number $\varepsilon > 0$, there is an integer K such that for all $i > K$,

$$d(a_i, a) < \varepsilon.$$

6. Suppose that a function $f : A \to B$ where A and B are CPOs is such that for all joinable sets $C \subseteq A$, $\hat{f}(C)$ is joinable and

$$\bigwedge \hat{f}(C) = f(\bigwedge C).$$

By itself, this does not imply that f is continuous, which is why the definition of a stable function in section **??** has to first assume that the function is continuous. To show that, we only need to find a function f that satisfies these conditions but is not continuous. Such a function is given by

$$f(a) = \begin{cases} \bot & \text{if } a \text{ is finite} \\ a & \text{otherwise} \end{cases}$$

(a) Show that this function is not continuous.

(b) Show that for all joinable sets $C \subseteq A$, $\hat{f}(C)$ is joinable and

$$\bigwedge \hat{f}(C) = f(\bigwedge C).$$

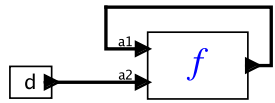
7. Consider an actor (called **followed by (FBY)** in the literature) with two input ports, *first* and *rest*, and one output port. Suppose that this actor first waits for an input on *first*, and when this arrives, it produces it on the output. After this, it henceforth waits for an input on *rest* and produces that on the output. However, while it is waiting for an input on *rest*, it can accept any input on *first*; it discards such an input. Show that this actor is sequential. Note, however, that this actor cannot be implemented with blocking reads.

8. This problem studies a determinate PN actor with behavior similar in some ways to the nondeterminate merge. Suppose D is a set of data values and $f\colon (D^{**})^2 \to D^{**}$ is a function defined by

$$f(s_1, s_2) = \underbrace{(e, e, \cdots, e)}_{N \text{ times}}$$

for all $s_1, s_2 \in D^{**}$, for some $e \in D$, and where $N = |s_1| + |s_2|$, the sum of the lengths of the sequences s_1 and s_2. If either input sequence is infinite, then the length of the output is infinite.

(a) Suppose this function is used in the following PN model:



where $d \in D^{**}$ is a finite sequence of length 5 produced by the left actor. What is the output of f?

(b) Is this function monotonic? Continuous?

(c) Is this function sequential?

9. This problem studies properties of the Scott topology over T^{**}.

(a) For a poset $(X, \leq)$ and subset $A \subseteq X$, an **upper set** $\uparrow A$ of A is defined to be

$$\uparrow A = \{x \in X \mid a \leq x \text{ for some } a \in A\}.$$

That is, an upper set of A is the set of all upper bounds of elements of A. Show that for the poset $(T^{**}, \sqsubseteq)$, the open sets of the Scott topology are upper sets.

(b) A **Hausdorff space** is a topology in which distinct points have disjoint neighborhoods. Specifically, if X is a set and τ is a topology over X, then for any two $x_1, x_2 \in X$ where $x_1 \neq x_2$, there are two open sets $X_1, X_2 \in \tau$ where $x_1 \in X_1$, $x_2 \in X_2$, and $X_1 \cap X_2 = \emptyset$. As a mnemonic, in a Hausdorff space, two distinct points can be "housed off" by open sets.

Show that the Scott topology over T^{**} is not a Hausdorff space. This property of the Scott topology accounts for much of the difficulty in reasoning about this topology, because the standard topology over the reals *is* Hausdorff.

(c) A **Kolmogorov space** is a topology in which given two distinct points, at least one of them has an open neighborhood not containing the other. Specifically, if X is a set and τ is a topology over X, then for any two $x_1, x_2 \in X$ where $x_1 \neq x_2$, there is an open set $A \subset X$ that contains only one of the two points x_1 and x_2. This condition intuitively means that the points are topologically distinguishable.

Show that the Scott topology over T^{**} is a Kolmogorov space.