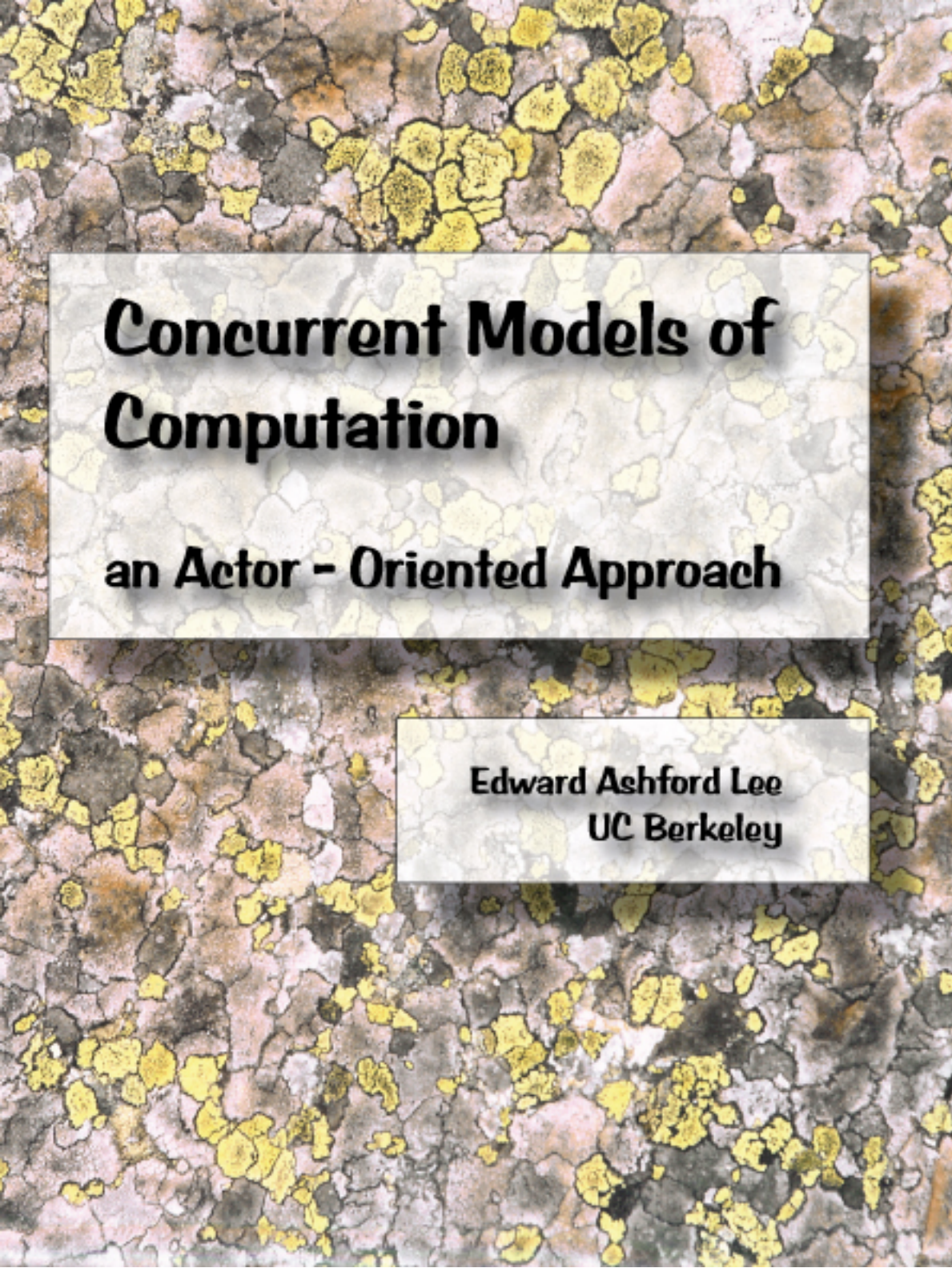

Concurrent Models of Computation
an Actor - Oriented Approach
Edward Ashford Lee
UC Berkeley

Zeroth Edition, Version 0.02- DRAFT
October 6, 2011

Please cite this book as:

Edward A. Lee,
Concurrent Models of Computation - An Actor-Oriented Approach,
http://Ptolemy.org/books, 2011.

Contents

Acknowledgements

The intellectual roots of this book lie in many of the best known minds in computer science and mathematics, including Alonzo Church, Gilles Kahn, Stephen Kleene, Alan Turing, and Dana Scott. I owe a particular debt to Gerard Berry, who showed me the beauty of the subject in 1994 during a sabbatical that I spent at INRIA in Sophia Antipolis, France.

My take on the subject of concurrent models of computation has also been greatly influenced by Albert Benveniste, Manfred Broy, Paul Caspi, Marc Geilen, David Harel, Tom Henzinger, Axel Jantsch, Hermann Kopetz, Leslie Lamport, Paul Le Guernic, Oded Maler, Florence Maraninchi, David Messerschmitt, Amir Pnueli, Alberto Sangiovanni-Vincentelli, Joseph Sifakis, Lothar Thiele, and Kees Vissers.

Several of my own former students and postdocs have also taught me a great deal and shaped my thinking on this subject, most particularly Joe Buck, Stephen Edwards, Johan Eker, Joern Janneck, Ben Lickly, Xiaojun Liu, Eleftherios Matsikoudis, Steve Neuendorffer, Tom Parks, John Reekie, Yang Zhao, and all participants in the Ptolemy project.

I gratefully acknowledge contributions and comments on early drafts of this text from the following people: Shaoyi Cheng, Aaron Culich, Ben Lickly, Xiaojun Liu, James Martin, Eleftherios Matsikoudis, Adam Megacz, Baruch Sterin, and Reinhard von Hanxleden.

1 Partially Ordered Sets

Contents

This chapter provides mathematical preliminaries used in subsequent chapters to develop the theory of concurrent systems. It reviews basic ideas and notation in logic, with particular emphasis on sets, functions, partial orders, and fixed-point theorems. The applications in subsequent chapters are essential to develop a full understanding of the role that these mathematical models play in concurrent systems.

1.1 Sets

In this section, we review the notation for sets. A **set** is a collection of objects. When object a is in set A, we write $a \in A$. We define the following sets:

- $\mathbb{B} = \{0, 1\}$, the set of **binary digits**.
- $\mathbb{N} = \{0, 1, 2, \cdots\}$, the set of **natural numbers**.
- $\mathbb{Z} = \{\cdots, -1, 0, 1, 2, \cdots\}$, the set of **integers**.
- $\mathbb{R}$, the set of **real numbers**.
- $\mathbb{R}_+$, the set of **non-negative real numbers**.

When set A is entirely contained by set B, we say that A is a **subset** of B and write $A \subseteq B$. For example, $\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$. The sets may be equal, so the statement $\mathbb{N} \subseteq \mathbb{N}$ is true, for example. The **powerset** of a set A is defined to be the set of all subsets. It is written $\wp(A)$ or 2^A (for a justification of the latter notation, see the sidebar on page 7). The **empty set**, written $\emptyset$, is always a member of the powerset, $\emptyset \in \wp(A)$.

We define **set subtraction** as follows,

$$A \backslash B = \{a \in A \ : \ a \notin B\}$$

for all sets A and B. This notation is read "the set of elements a from A such that a is not in B."

A **cartesian product** of sets A and B is a set written $A \times B$ and defined as follows,

$$A \times B = \{(a, b) \ : \ a \in A, b \in B\}.$$

A member of this set (a, b) is called a **tuple**. This notation is read "the set of tuples (a, b) such that a is in A and b is in B." A cartesian product can be formed with three or more sets, in which case the tuples have three or more elements. For example, we might write $(a, b, c) \in A \times B \times C$. A cartesian product of a set A with itself is written $A^2 = A \times A$. A cartesian product of a set A with itself n times, where $n \in \mathbb{N}$ is written A^n. A member of the set A^n is called an **n-tuple**. By convention, A^0 is a **singleton set**, or a set with exactly one element, regardless of the size of A. Specifically, we define $A^0 = \{\emptyset\}$. Note that A^0 is not itself the empty set. It is a singleton set containing the empty set (for insight into the rationale for this definition, see the box on page 7).

1.2 Relations and Functions

A **relation** from set A to set B is a subset of $A \times B$. A **partial function** f from set A to set B is a relation where $(a,b) \in f$ and $(a,b') \in f$ imply that $b = b'$. Such a partial function is written $f: A \rightharpoonup B$. A **total function** or just **function** f from A to B is a partial function where for all $a \in A$, there is a $b \in B$ such that $(a,b) \in f$. Such a function is written $f: A \to B$, and the set A is called its **domain** and the set B its **codomain**. Rather than writing $(a,b) \in f$, we can equivalently write $f(a) = b$.

Example 1.1: An example of a partial function is $f: \mathbb{R} \rightharpoonup \mathbb{R}$ defined by $f(x) = \sqrt{x}$ for all $x \in \mathbb{R}_+$. It is undefined for any $x < 0$ in its domain $\mathbb{R}$.

A partial function $f: A \rightharpoonup B$ may be defined by an **assignment rule**, as done in the above example, where an assignment rule simply explains how to obtain the value of $f(a)$ given $a \in A$. Alternatively, the function may be defined by its **graph**, which is a subset of $A \times B$.

Example 1.2: The same partial function from the previous example has the graph $f \subseteq \mathbb{R}^2$ given by

$$f = \{(x,y) \in \mathbb{R}^2 \ : \ x \geq 0 \text{ and } y = \sqrt{x}\} \ .$$

Note that we use the same notation f for the function and its graph when it is clear from context which we are talking about.

The set of all functions $f: A \to B$ is written $(A \to B)$ or B^A. The former notation is used when the exponential notation proves awkward. For a justification of the notation B^A, see the box on page 7.

The **function composition** of $f: A \to B$ and $g: B \to C$ is written $(g \circ f): A \to C$ and defined by

$$(g \circ f)(a) = g(f(a))$$

for any $a \in A$. Note that in the notation $(g \circ f)$, the function f is applied first. For a function $f: A \to A$, the composition with itself can be written $(f \circ f) = f^2$, or more

generally

$$\underbrace{(f \circ f \circ \cdots \circ f)}_{n \text{ times}} = f^n$$

for any $n \in \mathbb{N}$. In case $n = 1$, $f^1 = f$. For the special case $n = 0$, the function f^0 is by convention the **identity function**, so $f^0(a) = a$ for all $a \in A$. When the domain and codomain of a function are the same, i.e. $f \in A^A$, then $f^n \in A^A$ for all $n \in \mathbb{N}$.

For every function $f : A \to B$, there is an associated function $\hat{f} : \wp(A) \to \wp(B)$ defined on the powerset of A as follows,

$$\forall\, A' \subseteq A, \quad \hat{f}(A') = \{b \in B \ : \ \exists\, a \in A',\ f(a) = b\}.$$

We call $\hat{f}$ the **lifted** version of f. When there is no ambiguity, we may write the lifted version of f simply as f rather than $\hat{f}$ (see problem 4(c) for an example of a situation where there is ambiguity).

For any $A' \subseteq A$, $\hat{f}(A')$ is called the **image** of A' for the function f. The image $\hat{f}(A)$ of the domain is called the **range** of the function f.

Example 1.3: The image $\hat{f}(\mathbb{R})$ of the function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is $\mathbb{R}_+$.

A function $f : A \to B$ is **onto** (or **surjective**) if $\hat{f}(A) = B$. A function $f : A \to B$ is **one-to-one** (or **injective**) if for all $a, a' \in A$,

$$a \neq a' \Rightarrow f(a) \neq f(a'). \tag{1.1}$$

That is, no two distinct values in the domain yield the same values in the codomain. A function that is both one-to-one and onto is called a **bijection**.

Example 1.4: The function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = 2x$ is a bijection. The function $f : \mathbb{Z} \to \mathbb{Z}$ defined by $f(x) = 2x$ is one-to-one, but not onto. The function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = xy$ is onto but not one-to-one.

The previous example underscores the fact that an essential part of the definition of a function is its domain and codomain.

Proposition 1.1. *If $f: A \to B$ is onto, then there is a one-to-one function $h: B \to A$.*

Proof. Let h be defined by $h(b) = a$ where a is any element in A such that $f(a) = b$. There must always be at least one such element because f is onto. We can now show that h is one-to-one. To do this, consider any two elements $b, b' \in B$ where $b \neq b'$. We need to show that $h(b) \neq h(b')$. Assume to the contrary that $h(b) = h(b') = a$ for some $a \in A$. But then by the definition of h, $f(a) = b$ and $f(a) = b'$, which implies $b = b'$, a contradiction. □

The converse of this proposition is also easy to prove.

Proposition 1.2. *If $h: B \to A$ is one-to-one, then there is an onto function $f: A \to B$.*

Any bijection $f: A \to B$ has an **inverse** $f^{-1}: B \to A$ defined as follows,

$$f^{-1}(b) = a \in A \text{ such that } f(a) = b\ , \tag{1.2}$$

for all $b \in B$. This function is defined for all $b \in B$ because f is onto. And for each $b \in B$ there is a single unique $a \in A$ satisfying (1.2) because f is one-to-one. For any bijection f, its inverse is also a bijection.

1.2.1 Restriction and Projection

Given a function $f: A \to B$ and a subset $C \subseteq A$, we can define a new function $f|_C$ that is the **restriction** of f to C. It is defined so that for all $x \in C$, $f|_C(x) = f(x)$.

Example 1.5: The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is not one-to-one. But the function $f|_{\mathbb{R}_+}$ is.

Consider an n-tuple $a = (a_0, a_1, \cdots, a_{n-1}) \in A_0 \times A_1 \times \cdots \times A_{n-1}$. A **projection** of this n-tuple extracts elements of the tuple to create a new tuple. Specifically, let

$$I = (i_0, i_1, \cdots, i_m) \in \{0, 1, \cdots, n-1\}^m$$

for some $m \in \mathbb{N} \backslash \{0\}$. That is, I is an m-tuple of indexes. Then we define the projection of a onto I by

$$\pi_I(a) = (a_{i_0}, a_{i_1}, \cdots, a_{i_m}) \in A_{i_0} \times A_{i_1} \times \cdots \times A_{i_m} .$$

The projection may be used to permute elements of a tuple, to discard elements, or to repeat elements.

Projection of a tuple and restriction of a function are related. An n-tuple $a \in A^n$ where $a = (a_0, a_1, \cdots, a_{n-1})$ may be considered a function of the form $a\colon \{0, 1, \cdots, n-1\} \to A$, in which case $a(0) = a_0$, $a(1) = a_1$, etc. Projection is similar to restriction of this function, differing in that restriction, by itself, does not provide the ability to permute, repeat, or renumber elements. But conceptually, the operations are similar, as illustrated by the following example.

Example 1.6: Consider a 3-tuple $a = (a_0, a_1, a_2) \in A^3$. This is represented by the function $a\colon \{0, 1, 2\} \to A$. Let $I = \{1, 2\}$. The projection $b = \pi_I(a) = (a_1, a_2)$, which itself can be represented by a function $b\colon \{0, 1\} \to A$, where $b(0) = a_1$ and $b(1) = a_2$.

The restriction $a|_I$ is not exactly the same function as b, however. The domain of the first function is $\{1, 2\}$, whereas the domain of the second is $\{0, 1\}$. In particular, $a|_I(1) = b(0) = a_1$ and $a|_I(2) = b(1) = a_2$.

A projection may lifted just like ordinary functions. Given a set of n-tuples $B \subseteq A_0 \times A_1 \times \cdots \times A_{n-1}$ and an m-tuple of indexes $I \in \{0, 1, \cdots, n-1\}^m$, the **lifted projection** is

$$\hat{\pi}_I(B) = \{\pi_I(b) \ : \ b \in B\} .$$

1.3 Sequences

A tuple $(a_0, a_1) \in A^2$ can be interpreted as a sequence of length 2. The order of elements in the sequence matters, and is in fact captured by the natural ordering of the natural

Insight: Exponential Notation for Sets of Functions

The exponential notation B^A for the set of functions of form $f: A \to B$ is worth explaining. Recall that A^2 is the cartesian product of set A with itself, and that $\wp(A)$ is the powerset of A. These two notations are naturally thought of as sets of functions. A construction attributed to John von Neumann defines the natural numbers as follows,

$$\begin{aligned} \mathbf{0} &= \emptyset \\ \mathbf{1} &= \{\mathbf{0}\} = \{\emptyset\} \\ \mathbf{2} &= \{\mathbf{0}, \mathbf{1}\} = \{\emptyset, \{\emptyset\}\} \\ \mathbf{3} &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ \ldots & \end{aligned}$$

With this definition, the powerset $\mathbf{2}^A$ is the set of functions mapping the set A into the set $\mathbf{2}$. Consider one such function, $f \in \mathbf{2}^A$. For each $a \in A$, either $f(a) = \mathbf{0}$ or $f(a) = \mathbf{1}$. If we interpret "$\mathbf{0}$" to mean "nonmember" and "$\mathbf{1}$" to mean "member," then indeed the set of functions $\mathbf{2}^A$ represents the set of all subsets of A. Each such function defines a subset.

Similarly, the cartesian product $A^\mathbf{2}$ can be interpreted as the set of functions of form $f: \mathbf{2} \to A$, or using von Neumann's numbers, $f: \{\mathbf{0}, \mathbf{1}\} \to A$. Consider a tuple $a = (a_0, a_1) \in A^\mathbf{2}$. It is natural to associate with this tuple a function $a: \{\mathbf{0}, \mathbf{1}\} \to A$ where $a(\mathbf{0}) = a_0$ and $a(\mathbf{1}) = a_1$. The argument to the function is the index into the tuple. We can now interpret the set of functions B^A of form $f: A \to B$ as a set of tuples indexed by the set A instead of by the natural numbers.

Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ represent the set of **von Neumann numbers**. This set is closely related to the set $\mathbb{N}$ (see problem 4). Given a set A, it is now natural to interpret A^ω as the set of all infinite sequences of elements from A, the same as $A^\mathbb{N}$.

The singleton set $A^\mathbf{0}$ can now be interpreted as the set of all functions whose domain is the empty set and codomain is A. There is exactly one such function (no two such functions are distinguishable), and that function has an empty graph. Before, we defined $A^\mathbf{0} = \{\emptyset\}$. Using von Neumann numbers, $A^\mathbf{0} = \mathbf{1}$, corresponding nicely with the definition of a zero exponent on ordinary numbers. Moreover, you can think of $A^\mathbf{0} = \{\emptyset\}$ as the set of all functions with an empty graph.

It is customary in the literature to omit the bold face font for $A^\mathbf{0}$, $\mathbf{2}^A$, and $A^\mathbf{2}$, writing instead simply A^0, 2^A, and A^2.

numbers. The number 0 comes before the number 1. We can generalize this and recognize that a **sequence** of elements from set A of length n is an n-tuple in the set A^n. A^0 represents the set of empty sequences, a singleton set (there is only one empty sequence).

The set of all **finite sequences** of elements from the set A is written A^*, where we interpret $*$ as a wildcard that can take on any value in $\mathbb{N}$. A member of this set with length n is an n-tuple. The $*$ operator here is known as the **Kleene star** (or **Kleene operator** or **Kleene closure**), after American mathematician Stephen Cole Kleene (1909-1994).

The set of **infinite sequences** of elements from A is written $A^{\mathbb{N}}$ or A^{ω}. The set of **finite and infinite sequences** is written

$$A^{**} = A^* \cup A^{\mathbb{N}} .$$

Finite and infinite sequences play an important role in the semantics of concurrent programs. They can be used, for example, to represent streams of messages sent from one part of the program to another. Or they can represent successive assignments of values to a variable. For programs that terminate, finite sequences will be sufficient. For programs that do not terminate, we need infinite sequences.

1.4 Partial Orders

The notion of **order** is central to much of the structure in concurrent systems. Examples of ordering relationships include the usual intuitive ordering of numbers (0 is less than 1, etc.), but also many other concepts. For example, event a causes event b, 3.14159 is a better approximation of π than 3.14, and John is Sarah's father are all elements of ordering relations. For many such ordering relations, not all elements of the sets of interest are ordered. For example, John and Jane may not be related at all, unlike John and Sarah. Such partial orders play a central role in mathematical models of concurrent systems.

Definition 1.1. *A **partial order** on a set A is a relation from A to A satisfying the following properties. For all $a, b, c \in A$, the relation is*

1. ***reflexive**: $a \leq a$*
2. ***antisymmetric**: $a \leq b$ and $b \leq a$ implies that $a = b$.*
3. ***transitive**: $a \leq b$ and $b \leq c$ implies that $a \leq c$.*

In this definition, we have written the relation using the symbol $\leq$. Specifically, the relation is $\leq$, where $\leq \subseteq A \times A$, and $(a_0, a_1) \in \leq$ is equivalently written $a_0 \leq a_1$. The latter notation is referred to as **infix notation**.

A **partially ordered set** or **poset** is a set A and a partial order relation $\leq$ on that set. We can write a poset as a tuple $(A, \leq)$, or $(A, \leq_A)$ if we need to make a distinction between a partial order $\leq_A$ on set A and another partial order $\leq_B$ on another set B.

Example 1.7: A set A of sets and the subset relation $\subseteq$ form a poset $(A, \subseteq)$. The relation $\subseteq$ is referred to as the **subset order**.

For sets of sequences, there is a natural partial order relation based on the notion of a prefix.

Definition 1.2. *For a set A and the set of finite and infinite sequences A^{**} of elements of A, the **prefix order** is a relation $\sqsubseteq$ from A^{**} to A^{**} such that for any $s, s' \in A^{**}$, $s \sqsubseteq s'$ if either s is the empty sequence, or for all $n \in \mathbb{N}$ where $s(n)$ is defined, $s'(n)$ is defined and is equal to $s(n)$.*

It is easy to show that $(A, \sqsubseteq)$ is a poset by showing that the prefix order conforms with Definition 1.1.

Given a poset $(A, \leq)$ and two elements $a, a' \in A$, if either $a \leq a'$ or $a' \leq a$, then a and a' are said to be **comparable**. Otherwise, they are **incomparable**.

Definition 1.3. *A **chain** $C \subseteq A$ is a subset of a poset $(A, \leq)$ where any two members of the subset are comparable.*

A **total order** is a poset $(A, \leq)$ where A itself is a chain.

Example 1.8: The posets $(\mathbb{N}, \leq)$, $(\mathbb{Z}, \leq)$, and $(\mathbb{R}, \leq)$, where $\leq$ is the ordinary **numeric order** are total orders.

Example 1.9: The powerset $2^{\mathbb{N}}$ is the set of all sets of natural numbers. $(2^{\mathbb{N}}, \subseteq)$, where $\subseteq$ is the subset order, is a poset but not a total order.

Given a poset $(A, \leq)$, we can induce another relation $<$ called the **strict partial order** relation defined as follows.

$$\forall\, a, a' \in A, \quad a < a' \Leftrightarrow a \leq a' \text{ and } a \neq a' \,.$$

$(A, <)$ is called a **strict poset**.

1.4.1 Orders on Tuples

Given a poset $(A, \leq_A)$, the set A^2 has a **pointwise order** defined by

$$(a_0, a_1) \leq (a'_0, a'_1) \Leftrightarrow a_0 \leq_A a'_0 \text{ and } a_1 \leq_A a'_1,$$

for all $(a_0, a_1), (a'_0, a'_1) \in A^2$. With this order relation, $(A^2, \leq)$ is clearly a poset. An alternative order for A^2 is the **lexicographic order**, defined by

$$(a_0, a_1) \leq (a'_0, a'_1) \Leftrightarrow a_0 <_A a'_0 \text{ or } (a_0 = a'_0 \text{ and } a_1 \leq_A a'_1),$$

for all $(a_0, a_1), (a'_0, a'_1) \in A^2$.

Both pointwise order and lexicographic order generalize trivially to A^m for $m > 2$. They also generalize to A^*, $A^{\mathbb{N}}$, and A^{**}.

Example 1.10: Let $A = \{a, b, c, \cdots, z\}$ be the letters of the alphabet and $\leq$ be the usual **alphabetical order** for those letters. Then A^* is the set of all finite sequences of letters, which includes the set of all words. The lexicographic order for A^* is called the **dictionary order** because it gives the order in which words appear in a dictionary. The dictionary order is a total order, whereas the pointwise order for this poset would not be. In particular, under the pointwise order, the sequence (a, b) would be incomparable to the sequence (b, a), whereas under the dictionary order, the first is less than the second.

1.4.2 Upper and Lower Bounds

Given a poset $(A, \leq)$ and a subset $B \subseteq A$, an **upper bound** of B, if it exists, is an element $a \in A$ such that for all $b \in B$, $b \leq a$. A **least upper bound** or **LUB**, if it exists, is an upper bound a such that for all other upper bounds a' we have $a \leq a'$. A set may have an upper bound and no LUB.

Example 1.11: Consider the poset $(\mathbb{Q}, \leq)$ of **rational numbers**, where $\leq$ is the natural numeric order. Let $B \subset \mathbb{Q}$ be the subset of rational numbers whose value is less than the irrational number π (this relation "less than" is not the $\leq$ relation in this poset, since π is not a member of the poset, but we understand the definition of this subset anyway). In this poset, B has many upper bounds, but no least upper bound. Its LUB would have to be the greatest rational less than π, and there is no such greatest rational.

If a set $B \subseteq A$ has a least upper bound in the poset $(A, \leq)$, then it is said to be **joinable** in $(A, \leq)$, and the LUB is called the **join** of B and written $\bigvee B$.

Example 1.12: Consider the poset $(2^{\mathbb{N}}, \subseteq)$ of sets of natural numbers under the subset order. Every subset of $B \subseteq 2^{\mathbb{N}}$ is joinable and the join is the union of the sets in B,

$$\bigvee B = \bigcup_{b \in B} b \,.$$

It is easy to see that this is a LUB. Any other bound must contain at least this union. It is not accidental that the notation $\bigvee$ is similar to $\cup$. This similarity can be very helpful in getting used to the notation.

Correspondingly, a subset $B \in A$ may have a **lower bound** in the poset $(A, \leq)$. This will be an element $a \in A$ such that for all $b \in B$, $a \leq b$. The set B may also have a **greatest lower bound** or **GLB**, defined similarly to the LUB. The GLB, if it exists, is called the **meet** of B in $(A, \leq)$, and is written $\bigwedge B$.

Example 1.13: Any subset B as in the previous example has a meet given by

$$\bigwedge B = \bigcap_{b \in B} b \ .$$

Again, it is not accidental that the notation $\bigwedge$ reminds us of $\cap$.

1.4.3 Complete Partial Orders

A **pointed** poset $(A, \leq)$ is one that has a **bottom element**, written

$$\bot_A = \bigwedge A \in A \ .$$

The bottom element is less than or equal to every element in A. When the set is understood, the bottom element may be written simply $\bot$.

Definition 1.4. *A nonempty subset $D \subseteq A$ of poset $(A, \leq)$ is a **directed set** if every pair of elements in D has an upper bound in D. Equivalently, D is directed if every non-empty finite subset of D is joinable in D.*

Every chain is a directed set. In fact, directed sets can be viewed as generalizations of the idea of chains.

Definition 1.5. *A **complete partial order** or **CPO** $(A, \leq)$ is a pointed poset where every directed subset is joinable in A.*

CPOs have extremely useful properties and are widely used in computer science to study semantics. To understand why they are so useful, note first that any *finite* directed subset is trivially joinable, by the definition of a directed set. To be a CPO, this property has to extend to *infinite* directed subsets. This may seem like a small step, but it is not. Directed subsets in a CPO may be thought of as being "directed" towards some goal. That goal is the least upper bound, whose existence is guaranteed by the fact that the set is a CPO. This LUB is in a sense the "**limit**" of the directed set. This notion of a limit turns out to be very powerful and surprisingly widely applicable.

The intuition behind the LUB of a directed set is easiest to understand for a chain $C \subseteq A$, which is a particularly simple kind of directed set. In this case, interpreting the LUB as a

limit of the chain is completely natural. Being able to rely on the existence of this limit is valuable. Any ordered sequence in a CPO, therefore, has a limit. This is not a trivial property.

Example 1.14: None of the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ with numeric order is a CPO. Neither is the set of von Neumann numbers ω under the subset order. Each of these sets is itself a chain with no least upper bound, so clearly it cannot be that every chain has a least upper bound. If we augment $\mathbb{N}$ with an infinite element ∞, bigger than all elements in $\mathbb{N}$, then the resulting set $\mathbb{N} \cup \{\infty\}$ with the (extended) numeric order is a CPO. Similarly, the set $\omega \cup \{\omega\}$ is a CPO under the subset order, because every element of ω is a subset of ω.

The sets considered in the previous example are all totally ordered. For such sets, every directed subset is a chain. In fact, every subset is a chain. So to determine whether the set is a CPO, we only need to consider whether every chain has a least upper bound. Some sets that are not totally ordered also have the property that every directed set is chain.

Example 1.15: Given any set A, the set of finite sequences A^* of elements from A with the prefix order $\sqsubseteq$ is not a CPO. It is easy to construct a chain of growing finite sequences that has no upper bound that is itself a finite sequence. However, the set of finite and infinite sequences A^{**} is a CPO under the prefix order. To show this, first note that every directed subset of A^{**} is a chain (given two sequences that are both a prefix of a third, one of the two must be a prefix of the other). Thus, we simply note that given any chain, if the chain is finite, its least upper bound is its maximal element; if the chain is infinite, its least upper bound is the infinite sequence defined by the union of the elements of the chain. The notion of limits of such chains plays a major role in the semantics of concurrent systems.

It turns out that to decide whether any poset is a CPO, it is enough to consider only whether every chain has a least upper bound. It is not necessary to consider directed sets that are not chains. This follows from the following alternative definition of a CPO.

Definition 1.6. *A pointed poset $(A, \leq)$ is a **complete partial order (CPO)** if every chain in A is joinable in A.*

This definition captures the intuition every chain has a limit, in the sense of a least upper bound. To proof that this definition is equivalent to Definition 1.5 appears to be quite difficult. It seems to require the machinery of ordinals and the axiom of choice and is beyond the scope of this text. For a discussion, see Davey and Priestly (2002). In this text, we will use both definitions, though most of the time the second one will prove easier to use. In the case of the prefix order, every directed set is a chain, so the equivalence of the two definitions is trivial.

The following proposition enables us to construct more complicated CPOs from simpler ones.

Proposition 1.3. *Given a CPO $(A, \leq_A)$, $(A^n, \leq)$ is a CPO for any $n \in \mathbb{N}$, where $\leq$ is the pointwise order.*

Proof. First note that if $n = 0$, the proposition is trivial since the set has only one element, and every finite pointed poset is a CPO. It is also trivial for $n = 1$. For $n > 1$, we use definition 1.6 and consider only chains in A^n. Denote such a chain by

$$C = \{(a_{1,1}, a_{1,2}, \cdots, a_{1,n}), (a_{2,1}, a_{2,2}, \cdots, a_{2,n}), \cdots (a_{i,1}, a_{i,2}, \cdots, a_{i,n}), \cdots\}$$

where $i \leq j$ implies that $a_{i,m} \leq_A a_{j,m}$ for all $m \in \{1, \cdots, n\}$. Let $A_j = \{a_{1,j}, a_{2,j}, \cdots, a_{i,j}, \cdots\}$ for $j = 1, 2, \cdots$. It is easy to see then that

$$\bigvee C = (\bigvee A_1, \bigvee A_2, \cdots, \bigvee A_n) .$$

Hence, the chain has a LUB. Since every such chain has a LUB, the poset is a CPO. □

The following proposition follows trivially using the same proof technique, and proves quite useful.

Proposition 1.4. *Given a CPO $(A, \leq_A)$ and any set B, $(A^B, \leq)$ is a CPO, where $\leq$ is the pointwise order.*

The following example illustrates how we can pull together several of these ideas to begin addressing questions of program semantics.

Example 1.16: Let B be a set of variable names in a computer program and let A be the set of values that those variables can take on. During the (possibly nonterminating) execution of the program, a variable $b \in B$ takes on a (possibly infinite) sequence of values. The sequence of values is a member of the set A^{**}. We can define the **semantics** of the program to be a function $f: B \to A^{**}$ that for every $b \in B$ yields a sequence $f(b) \in A^{**}$. Since A^{**} is a CPO under the prefix order, by proposition 1.4 $(A^{**})^B$ is also a CPO under the **pointwise prefix order**.

Let a **partial execution** of the program be a function $f_i: B \to A^*$ for $i \in \mathbb{N}$. A partial execution always yields a finite sequence. If the execution of the program can be described as a chain of such partial executions (in the pointwise prefix order), then the limit of this chain in A^{**} can be taken to be the semantics f.

Describing an execution of program in this way is natural for many programs. Consider a determinate sequential program that updates values for variables as it executes. Each such update appends a new value to the end of the sequence consisting of the previous values of that variable. The sequence of growing sequences of values of variables is clearly a chain in the prefix order.

Any set can be turned into a CPO by choosing a **flat order relation**. Given an arbitrary set A, augment the set with one additional element that serves as the bottom element. Specifically, let B be the augmented set and the one additional element be $\perp_B$, so $B = A \cup \{\perp_B\}$. Define a poset $(B, \leq)$, where the order relation is such that $\perp_B \leq a$ for all $a \in A$ and $a \leq a' \Rightarrow a = a'$ for all $a, a' \in A$. This is called a **flat partial order**. Any two distinct elements a and a' in A are incomparable. Moreover, every directed subset of B is a chain with either one or two elements. Trivially, each such chain has an upper bound. Hence, $(B, \leq)$ is a CPO. In this CPO, all chains are finite, so the notion of a limit of a chain is rather trivial.

Example 1.17: An example of a flat partial order is illustrated in figure 1.1(a), where $A = \{a_1, a_2\}$ and $B = \{\perp_B, a_1, a_2\}$. Diagrams of that sort are known as **Hasse diagrams**. In a Hasse diagram, elements of the poset are placed above one another with line segments indicating the order relation. If two elements a and b are joined by a line segment, and a is below b in the diagram, then $a < b$.

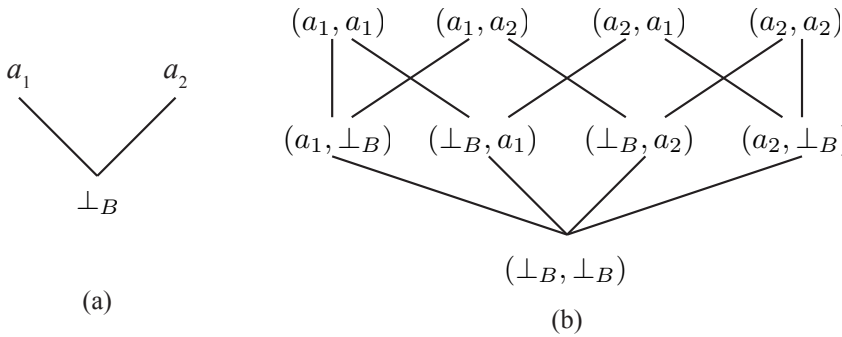


Figure 1.1: (a) Flat partial order for $B = \{\perp_B, a_1, a_2\}$. (b) Pointwise order on B^2.

Example 1.18: Suppose that $(B, \leq_B)$ is the flat partial order of the previous example. Then $(B^2, \leq)$, where $\leq$ is the pointwise order, is illustrated by the Hasse diagram in figure 1.1(b).

1.4.4 Lattices

Some partial orders have more structure than we have so far assumed.

Definition 1.7. *A **lattice** is a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique greatest lower bound $a \bigwedge a' \in A$ and a unique least upper bound $a \bigvee a' \in A$.*

Example 1.19: Given a set of sets A, the poset $(A, \subseteq)$ formed with the subset order is a lattice. Given two sets $a, a' \in A$, their GLB is their intersection $a \cap a'$, and their LUB is the union $a \cup a'$. If A is a powerset, then this lattice is known as the **powerset lattice**.

Example 1.20: The posets in figure 1.1 are not lattices. For example, in figure 1.1(a), the subset a_1, a_2 has no upper bound, much less a least upper bound. Adding an artificial top element (commonly denoted $\top$), would change these posets so that they both become lattices.

Some posets have some of the structure of lattices, but not all of it. A **lower semilattice** or **meet semilattice** is a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique greatest lower bound $a \bigwedge a' \in A$.

Example 1.21: The posets in figure 1.1 are lower semilattices. You can exhaustively verify that for any pair of elements, there is a single unique GLB.

Example 1.22: The poset $(A^{**}, \sqsubseteq)$ of finite and infinite sequences from the set A is a lower semilattice under the prefix order. Any two sequences in A^{**} have a unique common prefix (which ma y be the empty sequence).

An **upper semilattice** or **join semilattice** is dually defined as a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique least upper bound $a \bigvee a' \in A$.

A **complete lattice** is a poset $(A, \leq)$ where every subset (not just every pair elements) has a LUB and GLB (and similarly for **complete lower semilattice** or **complete upper semilattice**). A complete lattice is also a CPO.

Example 1.23: The powerset $2^{\mathbb{R}}$ of the reals with the subset order, $(2^{\mathbb{R}}, \subseteq)$ is a complete lattice. Given any set of sets of reals, the GLB is the intersection of the sets and the LUB is the union.

1.5 Functions on Posets

A function from a poset to a poset may preserve order.

Definition 1.8. *A function $f: A \to B$ from poset $(A, \leq_A)$ to poset $(B, \leq_B)$ is **monotonic** or **order preserving** if $a \leq_A a' \implies f(a) \leq_B f(a')$.*

Example 1.24: Consider a function $f: A^{**} \to A^{**}$ that is monotonic in the prefix order. This means that if a sequence s is a prefix of another s', i.e. that $s \sqsubseteq s'$, then $f(s) \sqsubseteq f(s')$. Functions with this property have many special qualities. The property implies that it is "safe" to evaluate the function with partial information about its input. Given only a prefix of what the input will eventually be, we can nonetheless evaluate the function using only the prefix, and we will get partial information (a prefix) about what the output will eventually be when the input is complete.

A function $f: A \to B$ is an **order embedding** if

$$a \leq_A a' \iff f(a) \leq_B f(a') \,. \tag{1.3}$$

Note that an order embedding is monotonic, but a monotonic function is not necessarily an order embedding. An order embedding is also necessarily one-to-one (see exercise 3). An order embedding $f: A \to B$ that is onto is called an **order isomorphism** from A to B. Since by exercise 3 it is also one-to-one, an order isomorphism is necessarily a bijection. Two posets are **order isomorphic** if there is an order isomorphism from one to the other. Order isomorphism is a rather strong relationship between posets. The posets are essentially the same, in that one can be obtained from the other by just renaming the elements.

Example 1.25: Let $A = \{0, 1, 2, \cdots, 255\} \subset \mathbb{N}$ and $B = \mathbb{B}^8$. Here, B is the set of all sequences of 8 binary digits. Let $f: B \to A$ be a function that for each $b \in B$ yields $f(b) \in A$ that is the number represented by b when b is interpreted as a binary unsigned number with the high-order bit first. Assume $\mathbb{B}$ is ordered so that $0 < 1$ and that B is endowed with a lexicographic order. Assume A is endowed with the usual numeric order. Then f is an order isomorphism.

If we are considering functions on CPOs rather than arbitrary posets, then an even more useful property than monotonicity is continuity.

Definition 1.9. *A function $f: A \to B$ from CPO $(A, \leq_A)$ to CPO $(B, \leq_B)$ is **continuous** if for all chains $C \subseteq A$,*

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

where $\hat{f}$ is the lifted version of f.

The **set of all continuous functions** from A to B is denoted $[A \to B]$, and is obviously a subset of the set B^A of all functions from A to B.

Proposition 1.5. *Every continuous function $f: A \to B$, where $(A, \leq_A)$ and $(B, \leq_B)$ are CPOs, is monotonic.*

Proof. Consider any $a, a' \in A$ where $a \leq a'$. Let $C = \{a, a'\}$, a chain. Note that $\bigvee C = a'$. Since f is continuous, we know that

$$\bigvee \hat{f}(C) = \bigvee \{f(a), f(a')\} = f(\bigvee C) = f(a') \,.$$

Hence, $f(a) \leq f(a')$, so the function is monotonic. □

Not every monotonic function is continuous however.

Example 1.26: Consider the CPO $(A, \leq)$, where $A = \mathbb{N} \cup \{\infty\}$ and $\leq$ is the numeric order. Let $f: A \to A$ be given by

$$f(a) = \begin{cases} 1 & \text{if } a \neq \infty \\ 2 & \text{otherwise} \end{cases}$$

This function is obviously monotonic. But it is not continuous. To see that, let $C = \mathbb{N}$ and note that $\bigvee C = \infty$. Hence, $f(\bigvee C) = 2$. However, $\hat{f}(C) = \{1\}$, because every element of C is finite. Hence, $\bigvee \hat{f}(C) = 1 \neq 2$. So the function is not continuous.

The following proposition specializes proposition 1.4 to continuous functions.

Proposition 1.6. *Given two CPOs $(A, \leq_A)$ and $(B, \leq_B)$, let $[B \to A] \subset A^B$ be the set of all continuous functions from B to A. Then $([B \to A], \leq)$ is a CPO under the pointwise order $\leq$.*

Proof. First we need to show that $[B \to A]$ has a bottom element. This is easy. The bottom element is a function $g \in [B \to A]$ where for all $b \in B$, $g(b) = \bot$. This function is obviously continuous and total, and hence is in $[B \to A]$.

Second, we need to show that any chain of functions in $[B \to A]$ has a LUB and that the LUB is continuous. Consider a chain of functions

$$C = \{f_1, f_2, ...\} \subset [B \to A].$$

Since each of these functions is continuous (and hence monotonic), then for any $b \in B$, the following set is also a chain,

$$C'_b = \{f_1(b), f_2(b), ...\} \subset A.$$

Since A is a CPO, this set has a LUB. Define the function $g\colon B \to A$ such that for all $b \in B$,

$$g(b) = \bigvee C'_b.$$

Then in the pointwise order, it must be that

$$g = \bigvee C = \bigvee \{f_1, f_2, ...\}.$$

It remains to show that g is in $[B \to A]$. To show this, we must show that it is continuous. We must show that for all chains $D \subset B$,

$$g(\bigvee D) = \bigvee \hat{g}(D).$$

Writing the elements of $D = \{d_1, d_2, ...\}$, observe that

$$\begin{aligned}
\bigvee \hat{g}(D) &= \bigvee \{g(d_1), g(d_2), ...\} \\
&= \bigvee \{\bigvee \{f_1(d_1), f_2(d_1), ...\}, \bigvee \{f_1(d_2), f_2(d_2), ...\} ...\} \\
&= \bigvee \{\bigvee \{f_1(d_1), f_1(d_2), ...\}, \bigvee \{f_2(d_1), f_2(d_2), ...\} ...\} \\
&= \bigvee \{\bigvee \hat{f}_1(D), \bigvee \hat{f}_2(D), ...\} \\
&= \bigvee \{f_1(\bigvee D), f_2(\bigvee D), ...\} \\
&= g(\bigvee D).
\end{aligned}$$

Note that the above implicitly uses the axiom of choice, which states that given a set of sets, one can construct a new set by collecting one element from each of the sets in the set of sets.

□

1.6 Fixed Points

Given a function $f: A \to A$, if there is value $a \in A$ such that $f(a) = a$, that value is called a **fixed point**. Existence and uniqueness of fixed point points play a central role in the semantics of programs. Existence of a fixed point will be interpreted as "the program has a meaning." Uniqueness will be interpreted as "the program has no more than one meaning." For functions that have multiple fixed points, we are often interested in the **least fixed point**, which is a fixed point $a \in A$ such that for any other fixed point $a' \in A$, $a \leq a'$. The following proposition assures the existence and uniqueness of such a least fixed point for continuous functions, and moreover gives a constructive procedure to determine that least fixed point.

Proposition 1.7. ***Kleene fixed-point theorem.*** *For any monotonic function $f: A \to A$ on CPO $(A, \leq)$, let*

$$C = \{f^n(\bot) \,:\, n \in \mathbb{N}\} \,.$$

Then if $\bigvee C = f(\bigvee C)$, $\bigvee C$ is the least fixed point of f. Moreover, if f is also continuous, then $\bigvee C = f(\bigvee C)$.

Proof. The first part of this theorem does not require that f be continuous, but only that it be monotonic. Suppose $\bigvee C = f(\bigvee C)$. This is a fixed point. Let a be any other fixed point, i.e. $f(a) = a$. We can show that $\bigvee C \leq a$, and hence $\bigvee C$ is the least fixed point. First, observe that $\bot \leq a$. Since f is monotonic, this implies that $f(\bot) \leq f(a) = a$. Again, since f is monotonic, this implies that $f(f(\bot)) \leq f(f(a)) = f(a) = a$. Continuing in this fashion, for any $n \in \mathbb{N}$,

$$f^n(\bot) \leq f^n(a) = a \,.$$

Hence, a is an upper bound of C. Since $\bigvee C$ is the least upper bound of C, it follows that $\bigvee C \leq a$, and hence $\bigvee C$ is the least fixed point of f.

For the second part of this theorem, we require that f be continuous, and not just monotonic. First, we observe that C is a chain in the CPO $(A, \leq)$. To see that, note that $\bot \leq f(\bot)$. Since f is monotonic, this implies that $f(\bot) \leq f(f(\bot))$. Continuing, we see that for all $n \in \mathbb{N}$, $f^n(\bot) \leq f^{n+1}(\bot)$, so C is a chain. Since C is a chain, it has a LUB $\bigvee C$.

Next note that $\hat{f}(C) \cup \{\bot\} = C$. Moreover, $\bigvee(\hat{f}(C) \cup \{\bot\}) = \bigvee \hat{f}(C)$ (an additional $\bot$ in a chain will not change its least upper bound). Combining these two facts, we conclude that $\bigvee \hat{f}(C) = \bigvee C$. But since f continuous, we also know that $\bigvee \hat{f}(C) = f(\bigvee C)$. Hence, $f(\bigvee C) = \bigvee C$, and hence $\bigvee C$ is a fixed point. □

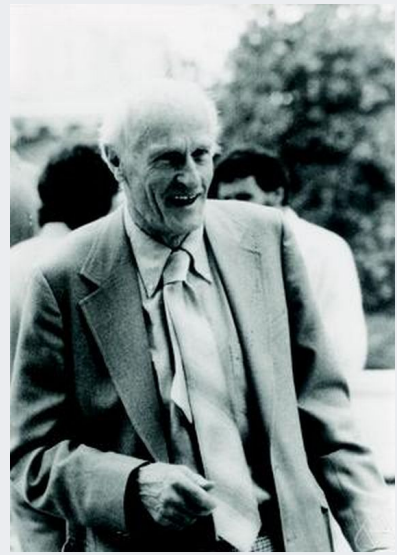
This theorem has profound consequences. It states that for a continuous function f, the least fixed point of this function can be found by first evaluating $f(\bot)$, then $f(f(\bot))$, then $f^3(\bot)$, etc. The result of this sequence of evaluations is a chain in A, and since A is a CPO, this chain has a limit. That limit is the least fixed point of the function. Applications of this theorem are scattered throughout the text.

Sidebar: Fixed Point Theorems

Proposition 1.7 is a variant of the **Kleene fixed-point theorem**, named after American mathematician Stephen Cole Kleene (1909–1994). The ascending chain $C = \{f^n(\bot) : n \in \mathbb{N}\}$ is known as the **Kleene chain** of the function f.

The Kleene fixed-point theorem is often attributed to Alfred Tarski, (1901–1983), a Polish-American logician and mathematician and a professor of mathematics at the University of California, Berkeley. **Tarski's fixed-point theorem** is similar to the Kleene fixed-point theorem, but in its original statement, it is about monotone functions on complete lattices. Because of this attribution, techniques used in computer science that are based on fixed-point theorems are often called **Tarskian**.

Another well-known theorem in this family is the **KnasterTarski fixed-point theorem**, developed earlier by Tarski and Bronislaw Knaster. This theorem is a special case of Tarski's fixed-point theorem that applies to the powerset lattice.



Stephen Cole Kleene (1909-1994). Photo by Konrad Jacobs, Erlangen, copyright (1978) MFO, Mathematisches Forschungsinstitut Oberwolfach, licensed under the Creative Commons Attribution-Share Alike 2.0 Germany license.

Exercises

1. This problem explores properties of onto and one-to-one functions.

   (a) Show that if $f: A \to B$ and $g: B \to C$ are onto, then $(g \circ f): A \to C$ is onto.

   (b) Show that if $f: A \to B$ is one-to-one and $g: B \to C$ is one-to-one, then $(g \circ f): A \to C$ is one-to-one.

2. Suppose A is some set and $S = A^{**}$ is the set of finite and infinite sequences of elements of A. This exercise explores some of the properties of the CPO S^n with the pointwise prefix order, for some non-negative integer n.

   (a) Show that any two elements $a, b \in S^n$ that have an upper bound have a least upper bound.

   (b) Let $U \subset S^n$ be such that no two distinct elements of U are joinable. Prove that for all $s \in S^n$ there is at most one $u \in U$ such that $u \sqsubseteq s$.

   (c) Given $s \in S^n$, suppose that $Q(s) \subset S^n$ is a joinable set where for all $q \in Q(s)$, $q \sqsubseteq s$. Then show that there is an s' such that $s = (\bigvee Q(s)).s'$, where the period operator specifies concatenation of sequences.

3. For two posets A and B:

   (a) Show that if $f: A \to B$ is an order embedding, then f is one-to-one.

   (b) Show that if $f: A \to B$ is an order isomorphism, then there is an order isomorphism $g: B \to A$.

4. Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ be the von Neumann numbers as defined on page 7. This problem explores the relationship between this set and $\mathbb{N}$.

   (a) Let $f: \omega \to \mathbb{N}$ be defined by

   $$f(x) = |x|, \quad \forall\, x \in \omega\,.$$

   That is, $f(x)$ is the size of the set x. Show that f is a bijection.

   (b) Let $(\omega, \subseteq)$ and $(\mathbb{N}, \leq)$ be posets, where $\subseteq$ is the usual subset order and $\leq$ is the usual numeric order. Show that these posets are order isomorphic.

   (c) The lifted version of the function f in part (a) is written $\hat{f}$. What is the value of $\hat{f}(\{\emptyset, \{\emptyset\}\})$? What is the value of $f(\{\emptyset, \{\emptyset\}\})$? Note that on page 4 it is noted that when there is no ambiguity, $\hat{f}$ may be written simply f. For this function, is there such ambiguity?

5. Classify each of the following as a partial order relation, a total order relation, or neither. State whether it is a CPO.

   (a) The set A = seconds, grams, meters, pounds, yards of units and a relation R defined by $(a, a') \in A$ if a can be converted to a' by scaling by a unitless constant.

   (b) The set D of all countable (including finite) subsets of $\mathbb{R}$, ordered by the subset order. **Hint:** It may be easier to use Definition 1.5 for a CPO, rather than Definition 1.6.

   (c) The set $\{1/n \mid n \in \mathbb{N}\} \cup \{0\}$, ordered by the natural numeric order (where $1/0$ is interpreted as ∞).

   (d) The set $\{1/n \mid n \in \mathbb{N}\}$, ordered by the reverse numeric order (where again $1/0$ is interpreted as ∞).

6. State whether each of the following posets is a CPO, and also where it is a lattice, lower semilattice, or upper semilattice. If you do not have enough information to determine whether this is a CPO or a lattice, then state this.

   (a) The set $A = [0,1) \subseteq \mathbb{R}$ under the numeric order, where $[0,1) = \{x \in \mathbb{R} : 0 \leq x < 1\}$.

   (b) The set $A = [0,1] \subseteq \mathbb{R}$ under the numeric order, where $[0,1] = \{x \in \mathbb{R} : 0 \leq x \leq 1\}$.

   (c) The set $A = \{x \in \mathbb{Q} : 0 \leq x \leq 1\}$ under the numeric order.

   (d) A pointed poset $(A, \leq)$ where every directed subset is finite.

   (e) A set A where every directed subset is a chain.

7. Assume two CPOs $(A, \leq)$ and $(B, \leq)$. Consider a poset $(A \times B, \leq)$ where the order is the lexicographic order.

   (a) Show that $(A \times B, \leq)$ is a CPO.

   (b) Suppose that $A = T_A^{**}$, $B = T_B^{**}$, and both CPOs use the prefix order, for arbitrary sets T_A and T_B. Suppose that $a \in T_A$ and $b_1, b_2 \in T_B$. Consider the set of sequences

   $$C = \{((a),(b_1)),((a,a),(b_2)),((a,a,a),(b_1)),...\}$$

   Under the lexicographic order, this is a chain. Find its LUB.

2 Process Networks

Contents

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

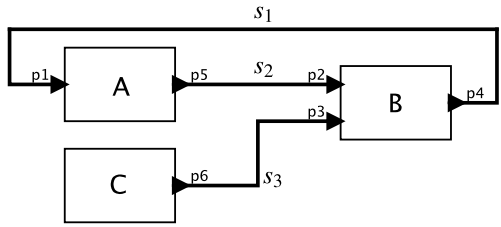


Figure 2.1: Example of a process network.

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 (2012).

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 2.1 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 2.1, 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 2.1, 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 2.1 executes the following Kahn-MacQueen procedure,

```
1 write(p5, 0);
2 while(true) {
3   t = read(p1);
4   write(p5, t);
5 }
```

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 2.1.

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 2.1 yields the following infinite 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 2.1 executes the following Kahn-MacQueen procedure,

```
1 while(true) {
2   t = read(p1);
3   write(p5, t);
4 }
```

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

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

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 2.1. 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.

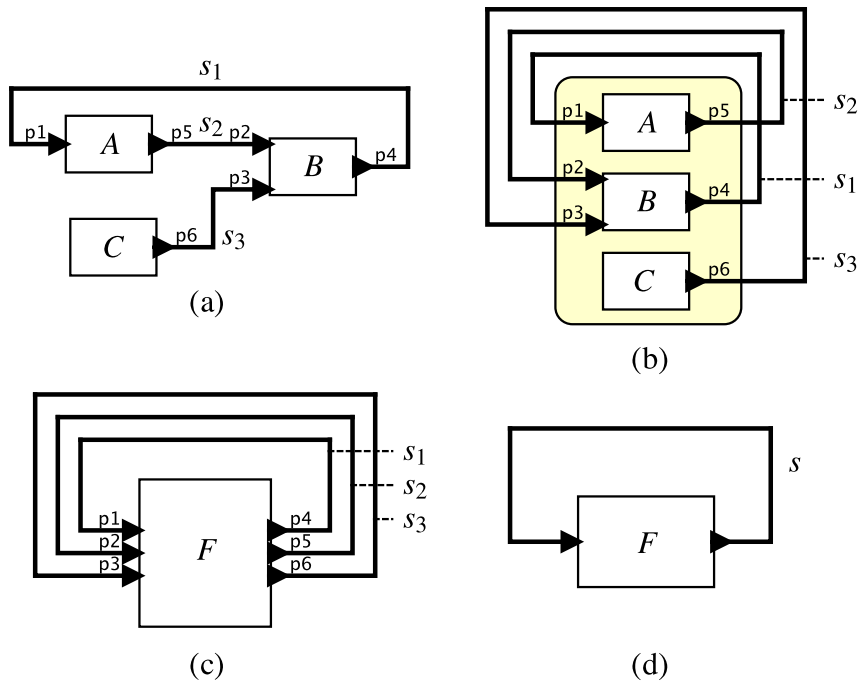


Figure 2.2: Execution of every process network can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

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

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

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

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

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 $F: \mathbb{N}^{**} \to \mathbb{N}^{**}$ 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).

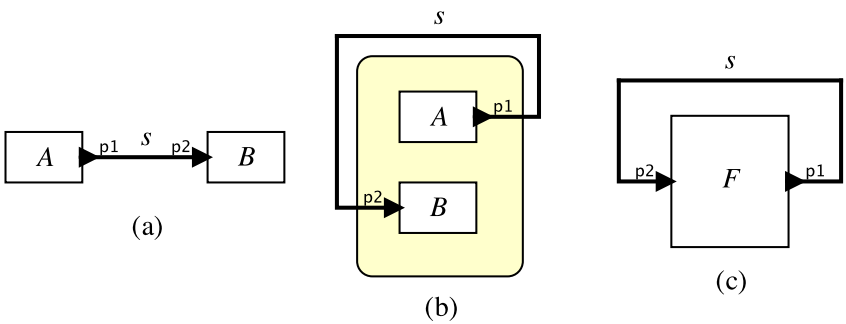


Figure 2.3: Even a process network with no feedback loops can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

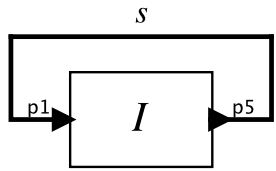


Figure 2.4: A process network with multiple fixed points, where I is the identity actor.

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 $A: T^{**} \to T^{**}$ 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 $F : (T^{**})^n \to (T^{**})^n$ is continuous if for all chains $C \subseteq (T^{**})^n$,

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

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 $\bigsqcup C = s$ 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 $\bigsqcup \hat{F}(C)$ 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,

```
1 while(true) {
2    read(p1);
3 }
4 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 $\bot$, and then recursively to the result, etc., as follows:

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

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

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

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

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

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 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 messages, 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.

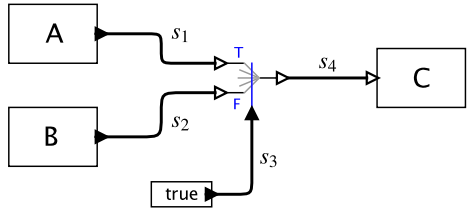


Figure 2.5: Example of a process network where unfair execution is desirable because fair execution will exhaust available memory.

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 $B = \{true, false\}$ 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,

```
1 while(true) {
2   c = read(control);
3   if (c) {
4     t = read(T);
5   } else {
6     t = read(F);
7   }
8   write(p, t);
9 }
```

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, and 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.

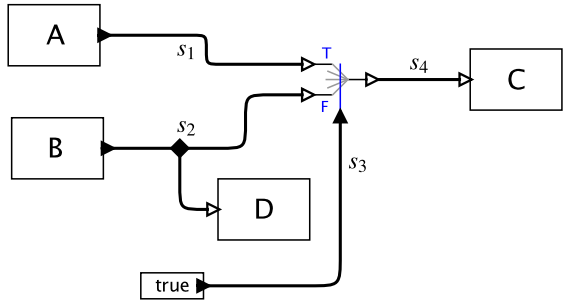


Figure 2.6: Example of a process network where demand-driven execution fails to deliver a bounded execution.

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

```
1 while (true) {
2     t = read(in);
3     write(out1, t);
4     write(out2, t);
5 }
```

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,

```
1 while (true) {
2   t = read(in);
3   write(out1, t);
4   write(out1, t);
5   write(out2, t);
6 }
```

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

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

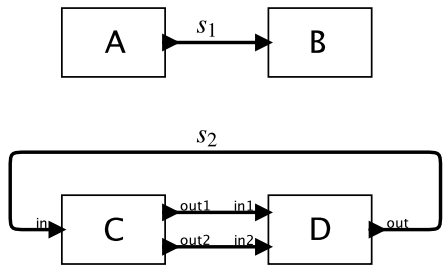


Figure 2.7: Example of a process network where Parks' algorithm does not yield an effective execution.

```
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 an **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 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,

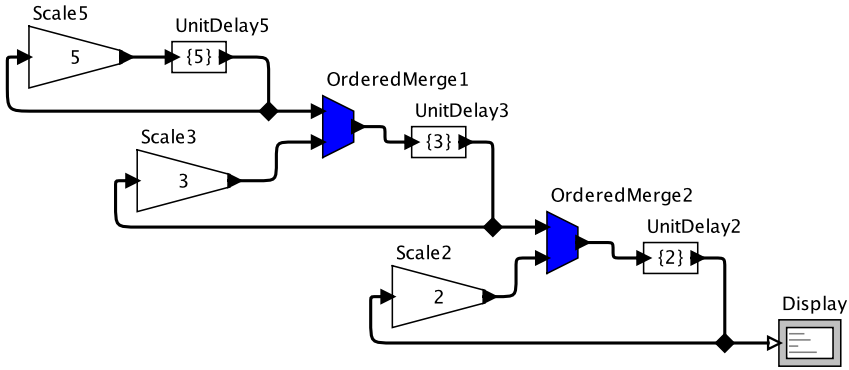


Figure 2.8: A process network for which it is difficult to determine appropriate buffer sizes.

actor B alone forms (trivial) undirected cycle, 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 $2^n3^m5^k$, 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 message with value N, then repeat forever the following sequence: read the input message and send it to the output. As before, the black diamonds are fork ac-

tors. They direct a single message 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 messages on each input port, they produce a numerically increasing merge of the two input sequences on their output ports (see Exercise **??** 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, at all times, finite, 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 family 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 "family 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 **??** for some properties of this toplogy.

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)

Bibliography

Berry, G., 1976: Bottom-up computation of recursive programs. *Revue Franaise dAutomatique, Informatique et Recherche Oprationnelle*, **10(3)**, 47–82.

Conway, M. E., 1963: Design of a separable transition-diagram compiler. *Communications of the ACM*, **6(7)**, 396–408.

Davey, B. A. and H. A. Priestly, 2002: *Introduction to Lattices and Order*. Cambridge University Press, second edition ed.

Dijkstra, E. W., 1976: *A Discipline of Programming*. Prentice Hall.

Faustini, A. A., 1982: An operational semantics for pure dataflow. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, vol. Lecture Notes in Computer Science (LNCS) Vol. 140, pp. 212–224.

Friedman, D. P. and D. S. Wise, 1976: CONS should not evaluate its arguments. In *Third Int. Colloquium on Automata, Languages, and Programming*, Edinburg University Press.

Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334.

Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co.

Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., pp. 993–998.

Landin, P. J., 1965: A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, **8(2)**, 89–101.

Morris, J. H. and P. Henderson, 1976: A lazy evaluator. In *Conference on the Principles of Programming Languages (POPL)*, ACM.

Parks, T. M., 1995: *Bounded Scheduling of Process Networks*. Phd thesis, UC Berkeley.

Ptolemaeus, C., ed., 2012: *System Design, Modeling, and Simulation Using Ptolemy II*. Ptolemy.org, Berkeley, CA, USA. Available from: http://ptolemy.org/books.

Ritchie, D. M. and K. L. Thompson, 1974: The UNIX time-sharing system. *Communications of the ACM*, **17(7)**, 365 – 375.

Stark, E. W., 1995: An algebra of dataflow networks. *Fundamenta Informaticae*, **22(1-2)**, 167–185.

Notation Index

Index