# On Determinism

Stephen A. Edwards

Columbia University in the City of New York

**Abstract.** The notion of deterministic execution of concurrent systems has appeared in many guises throughout Edward A. Lee's œuvre, but few really grasp how powerful, important, subtle, and flexible the concept really is. Determinism can be thought of as an abstraction boundary that delineates where control is passed from a system designer to the implementation. This paper surveys some of the many forms of determinism available in the models of computation Lee and others have proposed.

## 1 Introduction

Edward Lee and I have been chasing determinism for much of our careers, but the term means different things to different people. The dictionary definition of "determinism" is roughly the doctrine that nobody has free will, but our definition is more subtle. First, we concern ourselves with engineered systems instead of human beings[1]. Second, and more importantly, our notion of determinism actually permits a limited amount of free will—a system may make choices (both when it is implemented or when it is running) provided those choices do not affect the system's observed outputs. For example, we consider a combinational digital logic circuit to be deterministic even though the delays of its gates, and hence its detailed temporal behavior, may vary. The system specification only constrains the Boolean input/output relationship of the network; the physical behavior of an implementation of the network may vary provided the I/O relationship is respected.

In an attempt to more precisely characterize the notion of determinism, consider a quasi-formalism:[2] let $M = (S, I, O, C, E, B, p)$ be a *model of computation* (MoC) where $S$ is the set of all legal system specifications (i.e., supplied by a designer), $C$ be the set of all legal choices that can be made in implementing any system, $I$ and $O$ be the sets of inputs and outputs accounted for by the model of computation, $E$ and $B$ be the sets of environmental inputs and behaviors not accounted for by the model of computation, and $p : S \times C \to (I \times E \to O \times B)$ be the system implementation function for the model of computation, which takes

[1] At least I do; ignoring people is more-or-less why I entered engineering. That hasn't worked out too well.

[2] A quasi-formalism because the notions of these sets are far too abstract to be a proper formalism. In particular, the sets $E$ and $B$ are difficult to define because they are meant to represent "everything else," but this requires a careful definition of the universal set, which is not obvious.

a system specification and implementation choices and returns a system that transforms known and unaccounted-for inputs into known and unaccounted-for outputs. A model of computation $M$ is *deterministic* if for all $s \in S$, $c \in C$, $i \in I$, and $e \in E$, there is some function $d : S \times I \to O$ such that

$$p(s, c)(i, e) = \big(d(s, i), b\big). \tag{1}$$

In other words, the outputs that the model of computation accounts for *only* depend on the system specification and the inputs accounted for by the model of computation. Implementation choices and the environment may only affect the behavior of the system outside of these outputs.

By design, the above takes a very abstract view of what inputs and outputs, environmental or otherwise, may be. For example, inputs and outputs may be vectors of Boolean values, sequences of Boolean vectors over time, events tagged with timestamps [37], continuous-valued signals [38], and many more. In fact, a crucial choice in the design of a model of computation is whether such physical properties such as time, space, and voltage are considered part of a system's inputs and outputs versus being relegated to the environment. For example, in most classical models of computation in computer science (e.g., Turing machines), physical time is ignored; termination or the lack thereof was the only real concern. While such a view brings many theoretical benefits, it hinders the control of physical systems, which invariably depend strongly on time.

*Example 1.* Consider the model of computation embodied in an AND/INVERTER graph (AIG), a streamlined, abstract model of combinational Boolean logic networks proposed by Kuehlmann et al. [31] and used, for example, in Brayton and Mishchenko's ABC tool [10] to verify and synthesize digital logic circuits.

An AIG is a directed acyclic graph with three types of vertices: a vertex with two incoming arcs represents a logical AND gate; a primary input (i.e., from the environment) is modeled as a vertex with zero incoming arcs; and one particular vertex with no incoming arcs represents the constant "0." Vertices with a single or more than 2 incoming arcs are not allowed, but there is no constraint on the number of outgoing arcs from a vertex. Certain vertices are also considered outputs. Each arc has a Boolean inversion attribute that indicates whether the value flowing through it is to be complemented. For a particular assignment of input values to input vertices, the output from the network is an assignment of Boolean values to the output vertices that comes from an assignment of Boolean values to every AND vertex that satisfies all of them, i.e., each vertex takes on the logical AND of the values of the vertices along its incoming arcs, inverted according to the attribute on each arc.

It is easy to see such the output of such a network is deterministic. Since the graph is directed and acyclic, its vertices can be topologically ordered starting from the primary inputs, and the value of each vertex can be established in that order. The invariant is that a vertex's value is evaluated after its two fan-in vertices have been evaluated.

In this MoC, $S$ is the set of all AIGs; $I$ is an assignment of a Boolean value to each primary input vertex, and $O$ is the assignment of Boolean values to

each output vertex that is consistent with the inputs and the network. Choices $C$ that can be made during the implementation of the system include which logic gates to use, their speed, and how they are connected. Any circuit that ultimately gives the same input-output relationship is considered correct; its structure in not limited by the structure of the AIG. Environmental inputs might include fluctuations in supply voltage that could affect the delays of certain gates and noise coupled into the circuit from outside. The behavior $B$ may describe the voltages on each of the wires in the circuit as a function of time, or approximations to this, such as times at which the signals change.

The AIG MoC is deterministic in the sense of (1). An underlying assumption is that the choices $C$ are correct (i.e., produce a working circuit) and that the environmental inputs $E$ ultimately do not affect the output $O$.

*Example 2.* Consider the model of computation represented by the C programming language. In this MoC, $S$ is the set of all legal C programs; $C$ is the set of all choices a compiler may make during the compilation process, e.g., which instructions to choose, which registers to use, etc.

Defining $I$ and $O$, the inputs specified by the the model of computation, is a little subtle. $I$ includes command-line arguments, environment variables, the standard input stream, files in the filesystem, etc. $O$ includes the return value, the standard output stream, files the program writes to the filesystem, etc.

Defining $E$ and $B$ are more subtle still. $E$ can include things such as the type and speed of the processor in which the C program is being run, the time of day at which the program is run, and load and scheduling policy of the operating system under which it is run. $B$ includes things such as the time it takes to execute the program, the amount of power consumed by the computer while the program is running, and many others.

While programmers traditionally think of C as being deterministic, and most C programs behave deterministically once compiled, certain C constructs have unspecified behavior, meaning the C standard defines multiple possible behaviors but does not specify which must be chosen. Constructs may also have undefined behavior, meaning the standard imposes no requirements whatsoever, and implementation-defined behavior.

For example, C's argument evaluation order is unspecified. This readily leads to nondeterministic behavior when argument evaluation has interacting side-effects, such as in the (nonsensical) function call `foo(a=1, a=2)`. When the function `foo()` executes, the variable `a` will be either 1 or 2, but the C standard does not prescribe which (i.e., it is an implementation choice).

The C standard (e.g., ISO/IEC 9899:2011) attempts to legislate away the problems of nondeterminism by restricting the set of legal C programs $S$ to those that are *strictly conforming*: i.e., programs that do not produce output that depends on unspecified, undefined, or implementation-defined behavior.

Understandably, C programmers are taught to eschew unspecified, undefined, and implementation-defined behavior, but this approach is only partially effective. Although good C programmers are aware of and avoid such issues, in reality programmers rely on the C compiler at their disposal to test the legality of a

program and under this definition, the legality of a C program is technically undecidable. For example, while I was pleased to discover the version of GCC on my desktop machine (5.4) will produce a warning for the `foo(a=1, a=2)` example given the `-Wsequence-point` option, GCC failed to warn when the effects were moved to functions, i.e., `foo(one(), two())`.

Time is another thorny issue. The C standard provides the standard library function `time()` that returns the current calendar time. If programs that can call `time()` are part of $S$, the C MoC is deterministic only if $I$ includes the current time and fine details about the execution rate of the program.

A central tenet of determinism is that there are choices ($c$) to be made in the implementation of a system ($s$) that may affect its behavior ($b$), but they do not affect the output characterized by the model of computation.

I know of only a few mathematical approaches to determinism. Although there may be others, the deterministic MoCs I know of all use these. Below, I discuss these approaches and the models that use them.

## 2 The Banach Fixed-Point Theorem

Of the various fixed-point theorems at the root of deterministic MoCs, the Banach Fixed-Point Theorem is the easiest to state and understand. We start with a set (space) $X$ for which there is a *metric* $d : X \times X \to \mathbb{R}$ that represents a distance between two points $x, y \in X$, i.e., $d(x, x) = 0$, $d(x, y) > 0$ if $x \neq y$, $d(x, y) = d(y, x)$, and $d(x, y) \leq d(x, z) + d(z, y)$ (the triangle inequality).

**Theorem 1 (Banach [1]).** *If $X$ is a space with metric $d$ and $f : X \to X$ is a contraction mapping on $X$, i.e., there exists a Lipschitz constant $K < 1$ such that $d\big(f(x), f(y)\big) \leq Kd(x, y)$, then there is a unique fixed-point $x^*$, i.e., $f(x^*) = x^*$, where $x^* = \lim_{n\to\infty} f^n(x)$ for any $x \in X$.*

Two amazing things are happening here: that two points, after being mapped, come closer together is enough to ensure a unique fixed point, and that this fixed point can be found by starting anywhere and simply iterating. Intuitively, the Lipschitz constant $K$ provides a bound on how iterations of $f$ must behave, in particular telling us that they must grow closer.

In the MoC setting, the space $X$ is typically the output from the system, the mapping $f$ corresponds to taking some small step in running the system (e.g., evaluating a single logic gate), and the fixed point $x^*$ corresponds to a "stable" state of the system in which nothing more can or will be done to evaluate it. Determinism corresponds exactly to the fixed point being unique.

Lee et al. use Theorem 1 to show how discrete-event simulation models could be made deterministic. In such a model, a signal (i.e., communication history between processes) is modeled as a set of events—value-time pairs. The central challenge is choosing a suitable metric for what is otherwise a rather unwieldy space of possible behaviors $X$. Lee's earlier work [33] uses the Cantor metric $d(x, y) = 1/2^t$, where the time of each event is represented by a real number and

$t$ is the earliest time at which events in the two signals $x$ and $y$ differ. Later, Lee et al. [15, 16] adopt superdense time, in which each event is tagged by a real number-natural number pair to more delicately model simultaneous events. This complicates the metric, but Lee et al. show Theorem 1 can still be applied to establish determinism.

## 3 The Kleene/Knaster-Tarski Fixed-Point Theorem

The Banach Fixed-Point theorem relies on a metric that assigns real numbers to every pair of points in a space, which may be awkward in certain settings. Fortunately, another fixed-point theorem demands far less structure, making it easier to apply to MoCs. I state the theorem first then explain its details and implications.

**Theorem 2 (Kleene/Knaster-Tarski).** *Let $(X, \sqsubseteq)$ be a complete partial order with minimum element $\bot \in X$ and $f : X \to X$ a continuous function. $f$ has a unique least fixed point $\bigsqcup \{f^n(\bot) \mid n \in \{1, 2, \ldots\}\}$.*

Theorem 2, apparently a "folk theorem" variously attributed to Kleene and Knaster-Tarski [32, 53], instead of a metric, relies on a partial order relation, written $\sqsubseteq$ and sometimes pronounced "approximates," that is reflexive ($x \sqsubseteq x$), antisymmetric (if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$), and transitive (if $x \sqsubseteq y$ and $y \sqsubseteq z$, $x \sqsubseteq z$). The relation is partial because it may be the case that neither $x \sqsubseteq y$ nor $y \sqsubseteq x$, i.e., $x$ and $y$ may be incomparable. The usual subset relation $\subseteq$ is one such partial order. This mathematical machinery has been published in many places; Winskel [56] is my favorite; see also Scott [47] and Davey and Priestley [21].

Theorem 2 further requires the partial order to to be *complete*: any (increasing) chain $C = \{c_1, c_2, \ldots\}$ (where $c_1 \sqsubseteq c_2 \sqsubseteq \cdots$) must have a least upper bound $\bigsqcup C \in D$ satisfying $c_k \sqsubseteq \bigsqcup C$ (i.e., $\bigsqcup C$ is an upper bound) and $\bigsqcup C \sqsubseteq b$ for any $b$ such that $c_k \sqsubseteq b$ (i.e., $\bigsqcup C$ is the least such bound). Intuitively, increasing sequences in the space can not increase forever.

Finally, Theorem 2 requires a *continuous* function (sometimes termed "Scott continuous" after Dana Scott [46], who pioneered their use for modeling recursion in denotational semantics). Continuity is analogous to the usual definition for real-valued functions: the limit of the function is the function at the limit, i.e., for all chains $C$, $\bigsqcup\{f(c) \mid c \in C\} = f(\bigsqcup C)$. Informally, nothing strange happens when you actually reach a limit. Moreover, continuity implies monotonicity, i.e., $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

The sketch of the proof of Theorem 2 is quick and illuminating. Montonicity implies $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots$ is a chain. Because $\sqsubseteq$ is complete, this chain $C$ has a unique least upper bound $\bigsqcup C$. Finally, because $f$ is continuous, $\bigsqcup C$ is a fixed point because $f(\bigsqcup C) = \bigsqcup\{f(c) \mid c \in C\} = \bigsqcup C$.

Put another way, iterating $f$ produces a nondecreasing sequence that approaches a unique least upper bound, which happens to be the least fixed point.

Theorem 2 only guarantees a unique least fixed point; $f$ may have other, greater fixed points.

Perhaps most famously, Kahn [30] uses Theorem 2 to show his process networks are deterministic. Kahn networks consist of sequential processes that communicate through unbounded FIFO channels. Each process may compute, emit a token to an output channel, or wait for the next token on an input channel. Kahn models the contents of each channel as $D^\omega$: the set of finite and infinite sequences over the set of tokens $D$, and the domain $X$ is a vector of channels. Kahn shows that each process behaves as a continuous function under his restrictions, e.g., monotonicity follows from blocking reads: additional input tokens can never make a process "unemit" tokens or "change its mind" about a token emitted earlier; continuity follows because a process can't "wait forever" before generating an output. Kahn defines the behavior of his networks as the least fixed point of the function composed from the functions of all the processes, which is therefore unique. Furthermore, the proof of Theorem 2 tells us that this fixed-point may be reached (or at least approximated) by simply running the processes.

In the vocabulary of (1), Kahn networks have $S$ as the Kahn network, $I$ and $O$ are the sequences of tokens on the channels, $C$ and $E$ include implementation choices, e.g., with respect to scheduling the execution of the processes, and $B$ includes the timing of the tokens on the channels.

Kahn's networks and its underlying mathematics have spawned a host of variants. Lee's Synchronous Dataflow (SDF) [35] is a restriction of Kahn networks to regular, statically known communication patterns, thus piggybacks on Kahn's result to guarantee determinism. Many slight variants have been proposed, including cyclo-static dataflow [8] and Boolean dataflow [14]. Lee and Parks [36] discuss many of these models. Lee and Matsikoudis [34] show how dataflow actors with firing rules behave like Kahn processes (i.e., continuous functions over streams). My own SHIM formalism [24, 22] falls somewhere between the rigid, predictable communication patterns of Lee's SDF and Kahn's Turing-complete process networks by restricting processes communicate via rendezvous to bound buffer sizes. Lately, I have devised yet another deterministic dataflow formalism derived from Kahn, this time synthesizing deterministic hardware from bounded-buffer dataflow networks [25].

Kahn relies on the ability of Theorem 2 to cope with infinite domains, but finite domains often suffice.

For example, Theorem 2 also provides determinism to cyclic combinational logic circuits and related block diagram languages. In classical three-valued circuit simulation, the domain $X$ is a finite vector of finite elements: three-valued wire values where the unknown value (usually written "X" in the engineering literature) is the least element $\perp$ and $\perp \sqsubseteq 0$ and $\perp \sqsubseteq 1$ where 0 and 1 are incomparable.

Three-valued digital logic simulation has been around since at least the 1950s. Muller [43] was one of the earliest to consider it in light of the works of Kleene and others. Eichelberger [26] showed how to use it to detect switching hazards

in circuits. Bryant [11] used this logic to simulate switching networks built from MOS transistors that could include such oddities as pass gates and dynamic logic families. Later, researchers including Brzozowski and Seger [13, 12], Malik [40] and Shiple and Berry [48] connected three-valued simulation to the analysis of logic circuits with loops and time models, ultimately showing it is a precise abstraction of logic gate networks with unknown timing [42].

Berry [5] adopted what is essentially three-valued logic simulation semantics for later versions of his Esterel language [6] to resolve some longstanding questions about which programs were self-contradictory. He also noted the connection between three-valued simulation, Theorem 2, and constructive logic, dubbing this treatment the constructive semantics of Esterel [4].

My own thesis work, which Lee oversaw, produced a block-diagram language [23] whose deterministic semantics amounted to three-valued simulation abstracted further to allow general monotonic functions to operate on arbitrary data, not just Boolean. August and his group at Princeton used this approach in their Liberty processor simulation environment [44]. More recently, Lee and Zheng [39] sewed this model together with discrete-event simulation.

Theorem 2 is often applied in a setting where the behavior and/or implementation of a system may be one of a family of functions $f$ that arise from evaluating parts of a system at different rates. For example, implementing an SDF graph usually involves scheduling the rates and execution order of the processes, which generally affects the function $f$ [7]. Fortunately, it turns out that such restructuring does not affect the fixed point. Bekić [3] shows, for example, that a system may be split apart and the parts run asynchronously but their results ultimately merged without affecting the fixed point. See also Winskel [56, Ch. 10].

Such an asynchronous approach to computing a function is usually termed "chaotic iteration," and is a common way to compute large functions on parallel hardware. Cousot and Cousot [20] and Wei [55] observe the connection between this approach and Theorem 2. Bourdoncle [9] shows how wisely partitioning the graph of a system can reduce the amount of effort involved in evaluating it without affecting the result.

## 4 Church-Rosser, Confluence, and the Lambda Calculus

Church's lambda calculus [17, 18, 2] is a remarkable piece of mathematics in that it is deceptively simple yet somehow all-encompassing. The basis of functional programming including McCarthy's LISP [41], Sussman and Steele's Scheme [50], Milner's ML [27], and Haskell [28], it reduces computation to little more than substituting arguments for variables in functions, which, amazingly, is enough to make it as powerful as Turing machines [54]. Expositions of the lambda calculus abound. Berendregt [2] is the all-inclusive reference, but I much prefer Peyton Jones [29, Ch. 2] as a place to start. Stoy [49] also provides a readable treatment.

Unlike Theorems 1 and 2, the lambda calculus only guarantees that a fixed point is unique *if it exists*. This is a side-effect of the "batch mode" bias in the

lambda calculus: it was intended to model computation that produces a result only when it terminates.

Another big difference of the lambda calculus compared to Theorems 1 and 2 is its explicit use of choice in the evaluation "function." The lambda calculus proceeds not by applying a particular function $f$, but by applying a rewriting procedure that may make choices that produce different (intermediate) results.

A *lambda expression* is either $x$ (a variable), $(\lambda x.M)$ (a lambda abstraction—a model of a function), or $(M\,N)$ (application of the expression $M$ to argument $N$), where $M$, $N$, ... are lambda expression and $x$, $y$, ... are variables.

For example, $(\lambda x.x)$ represents the identity function; $(\lambda x.(\lambda y.x))$ is a function that takes an argument $x$ and returns a function that takes an argument $y$, ignores it, and returns $x$, which can be used to represent the Boolean "true." To improve the readability of lambda expressions, parentheses are dropped where ambiguity can be resolved by taking the body of a lambda abstraction to extend as far to the right as possible and taking juxtaposition as associating left-to-right, e.g., $(\lambda x\,.\,x\,y\,z)\,w$ means $((\lambda x\,.\,((x\,y)\,z)))\,w)$.

A *reducible expression* or *redex* is a lambda expression of the form $((\lambda x\,.\,M)\,N)$, i.e., where a lambda abstraction is being applied and thus computation is to be performed. For example, $(\lambda z.z)\,y$ is a redex in which the identity function is being applied to $y$, but $(\lambda x\,.\,x)$, $x\,y$, and $x\,(\lambda x\,.\,x)$ are not redexes.

The one interesting computational step in the lambda calculus is $\beta$-reduction, in which a redex is replaced with a version of the body of the lambda abstraction in which every instance of the variable is replaced with the argument:

$$((\lambda x\,.\,M)\,N) \to_\beta M[x := N] \quad (\beta)$$

where $M[x := N]$ means a copy of $M$ in which all free[3] instances of $x$ have been replaced with the argument $N$. So for example, $(\lambda x\,.\,\lambda y\,.\,x)\;(\lambda z\,.\,z) \to_\beta \lambda y\,.\,\lambda z\,.\,z$.

In general, $\beta$-reduction can be applied anywhere in a lambda expression, not just at the top level as prescribed by the $(\beta)$ rule. To do this, the $\to_\beta$ rule is extended with three others that allow $\beta$-reduction to be performed inside the body of a lambda abstraction, or on either the left or right side of an application:

$$\frac{M \to_\beta M'}{(\lambda x\,.\,M) \to_\beta (\lambda x\,.\,M')}\,(\text{body})$$

$$\frac{M \to_\beta M'}{(M\,N) \to_\beta (M'\,N)}\,(\text{left}) \qquad \frac{N \to_\beta N'}{(M\,N) \to_\beta (M\,N')}\,(\text{right})$$

In general, the $(\beta)$, (left), and (right) rules may each apply to a lambda expression, which introduces choice. For example, applying $(\beta)$ to $(\lambda x\,.\,\lambda y\,.\,y)\,((\lambda w\,.\,w\,w)\,(\lambda z\,.\,z\,z))$ produces $\lambda y\,.\,y$ since $x$ does not appear in the body of the $\lambda x$ expression. However, the (right) rule also applies to this expression, which

[3] I am sidestepping all the fussy bookkeeping necessary to deal with reused variable names because it is ultimately bland, mathematically speaking. See, e.g., Peyton Jones [29, Ch. 2].

allows the argument to the $\lambda x$ expression to be reduced before it is substituted, giving

$$\frac{(\lambda w \,.\, w\, w)\,(\lambda z \,.\, z\, z) \rightarrow_\beta (\lambda z \,.\, z\, z)\,(\lambda z \,.\, z\, z)}{(\lambda x \,.\, \lambda y \,.\, y)\,((\lambda w \,.\, w\, w)\,(\lambda z \,.\, z\, z)) \rightarrow_\beta (\lambda x \,.\, \lambda y \,.\, y)\,((\lambda z \,.\, z\, z)\,(\lambda z \,.\, z\, z))}\,.$$

A model of computation in which a choice may be taken is at the heart of nondeterminism. Superficially, it would seem that allowing a model to take different steps that produces different results (as the above example illustrated) would produce a nondeterministic model, but this turns out not to be the case for the lambda calculus, as Church and Rosser originally showed.

As defined above, determinism constrains the relationship between the inputs and outputs of a model of computation, but not choices and behavior of how the system implements the I/O relationship. In the lambda calculus, a redex represents work yet to be done, i.e., a function that can still be evaluated; any expression that contains a redex is not (yet) at the point where it will generate an output.

A lambda expression is in *normal form* if it contains no redex, i.e., if $\beta$-reduction cannot be applied. "Execution" of a lambda expression amounts to applying $\beta$-reduction (i.e., $\rightarrow_\beta$) until the expression reaches normal form, which is considered the "output" of a lambda expression.

It turns out the lambda calculus is deterministic due to a remarkable result by Church-Rosser: if a lambda expression can be $\beta$-reduced into normal form, there is only one such normal form. In other words, making choices about which redex to reduce cannot affect the ultimate result. There are lambda expressions that do not have a normal form, perhaps the simplest of which is $(\lambda x \,.\, x\; x)\;(\lambda y \,.\, y\; y)$ ($\beta$-reduction can be applied indefinitely yet the expression does not effectively change). These are analogous to non-terminating programs on, say, Turing machines.

The proof of determinism works in two steps. First, $\beta$-reduction is *confluent*:

**Theorem 3 (Church-Rosser [19]).** *Let $\rightarrow_{\beta^*}$ represent one or more applications of the $\rightarrow_\beta$ relation. If $M \rightarrow_{\beta^*} N_1$ and $M \rightarrow_{\beta^*} N_2$, then there exists an $M'$ such that $N_1 \rightarrow_{\beta^*} M'$ and $N_2 \rightarrow_{\beta^*} M'$.*

The second, easier step observes confluence implies an expression's normal form, if any, is unique:

**Corollary 1.** *No lambda expression can be $\beta$-reduced to two different normal forms.*

*Proof.* Suppose $M \rightarrow_{\beta^*} N_1$, $M \rightarrow_{\beta^*} N_2$, and both $N_1$ and $N_2$ are in normal form. By Theorem 3, this means there exists $M'$ such that $N_1 \rightarrow_{\beta^*} M'$ and $N_2 \rightarrow_{\beta^*} M'$. However, since both $N_1$ and $N_2$ are in normal form, they contain no redexes, so it must be that $N_1 = N_2$.

The proof of Theorem 3 is not obvious because $\beta$-reduction can substantially restructure an expression. Reducing a redex may make others disappear, e.g.,

since $(\lambda x\,.\,y)\ M \to_\beta y$, any redex in $M$ goes away. Reducing a redex may also make copies of a redex, e.g., since $(\lambda x\,.\,x\ x\ x)\ M \to_\beta M\ M\ M$, any redex in $M$ is copied three times. "Reducing" the lambda expression $(\lambda x\,.\,x\ x\ x)\ (\lambda y\,.\,y\ y\ y)$ actually makes it increase without bound. In general, this makes $\beta$-reduction non-monotonic, precluding a proof like that for Theorem 2.

The usual proof of Theorem 3 demonstrates confluence by showing it is possible, after any single $\beta$-reduction of a redex, to reach the configuration obtained by reducing *all* redexes "in parallel." Induction on this step completes the proof, which Tait and Martin-Löf developed in the early 1970s but did not publish. Barendregt [2] recites this proof and others, but I prefer Pollack's [45] treatment of Takahashi's presentation [51, 52].

The rules for maximal parallel $\beta$-reduction are deterministic:

$$x \Rightarrow x \quad \text{(p-var)} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{(\lambda x\,.\,M)\ N \Rightarrow M'[x := N']}\ \text{(p-}\beta\text{)}$$

$$\frac{M \Rightarrow M'}{\lambda x\,.\,M \Rightarrow \lambda x\,.\,M'}\ \text{(p-}\lambda\text{)} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N' \quad M \text{ is not a lambda}}{M\ N \Rightarrow M'\ N'}\ \text{(p-app)}$$

The (p-var) rule is the base case, which leaves unbound variables unchanged. The (p-$\lambda$) rule handles a lambda term by rewriting its body. Finally, (p-$\beta$) and (p-app) handle applications. The (p-$\beta$) rule performs the usual $\beta$-reduction on redexes, but only *after* reducing all redexes in the body $M$ and the argument $N$. The (p-app) rule applies to every other application term (e.g., when $M$ is an application or a variable) and reduces all redexes in both of its sub-expressions.

However, reducing all redexes according to these deterministic, maximally parallel rules does not necessarily produce a normal form; reductions may expose new ones that were not initially "visible" to the parallel $\beta$-reduction rules. For example, reducing $((\lambda w\,.\,\lambda x\,.\,w)\ y)\ z$ to normal form takes two steps. The first:

$$\dfrac{\dfrac{\dfrac{w \Rightarrow w}{\lambda x\,.\,w \Rightarrow \lambda x\,.\,w}\ \text{(p-}\lambda\text{)} \qquad y \Rightarrow y}{(\lambda w\,.\,(\lambda x\,.\,w))\ y \Rightarrow \lambda x\,.\,y}\ \text{(p-}\beta\text{)} \qquad z \Rightarrow z}{((\lambda w\,.\,(\lambda x\,.\,w))\ y)\ z \Rightarrow (\lambda x\,.\,y)\ z}\ \text{(p-app)}$$

Determinism in the lambda calculus, therefore, has parallels with acyclic digital electronic logic circuits: the implementation may choose to do more work than necessary, but the outcome of needless work does not affect the ultimate result. In a lambda expression, "more work" would be performing $\beta$-reductions on terms that are eventually ignored. Similarly, a circuit may "glitch" and transition more than necessary because of multiple paths with different delays to a particular logic gate. However, because an acyclic circuit is finite and contains finitely many paths, glitching always converges, whereas it impossible in general to guarantee $\beta$-reduction will converge because the model is Turing-complete.

For practical reasons, most functional languages (e.g., LISP, Scheme, and ML) have adopted the applicative execution policy familiar to most programmers, i.e., function arguments are evaluated before the function is invoked. In the lambda

calculus, this corresponds to reducing the argument to a lambda term to normal form before performing $\beta$-reduction. However, other languages, notably Haskell, adopts a more lazy strategy in which evaluation is deferred. The result is that certain programs coded in Haskell (e.g., those that manipulate infinite lists) will terminate while the same programs in other functional languages do not. However, if a Haskell program is coded in an applicative function language and still terminates, Church-Rosser ensures the result is the same.

## 5 Conclusion

I presented a very abstract model of models of computation that gives us a starting point for speaking about the determinism of MoCs, provided some examples and their relationship to the model, and discussed three well-known theorems that provide determinism to many models of computation.

The goal of determinism is to provide implementation flexibility (e.g., to optimize metrics such as speed or cost) without these choices affecting how a designer understands the behavior of the system. Technically speaking, I define a deterministic model of computation as one in which the relationship between the defined inputs and outputs of the system is a function that is unaffected by choices made during its implementation or operation.

The Banach Fixed-point Theorem (Theorem 1) shows a contracting function in a metric space converges to a fixed point. Lee et al. used this for arguing the determinism of certain discrete-event models.

The Kleene-Knaster-Tarski Theorem (Theorem 2) relies on a partial order with well-defined limits and a continuous function, which also happens to be monotonic. In this setting, iterations starting from the least defined element $\perp$ in the space converge to a unique least fixed point. Kahn [30] used this to show his process networks were deterministic and many variants since then, including Lee's SDF [35], have also relied on this result to ensure a parallel, asynchronous implementation of a system remains deterministic.

The lambda calculus has the Church-Rosser Theorem (Theorem 3), which states the ultimate result of computation (the normal form of a lambda expression) is unique if it exists. Reducing an expression to normal form involves making choices (either statically, as part of the implementation choices, or dynamically), but Church-Rosser says these choices only affect performance, not the ultimate result. This theorem provides determinism guarantees to many functional languages and can also be used in a parallel setting.

It is my hope that this survey has clarified your understanding of the meaning and utility of determinism in MoCs, perhaps providing inspiration of how to ensure determinism in the next MoC you devise.

## Acknowledgements

The National Science Foundation funded this work (CCF-1162124); the suggestions of two anonymous reviewers definitely improved this paper.

## References


1. Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3(1):133–181, 1922.
2. Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.
3. H. Bekić. Definable operations in general algebras, and the theory of automata and flowcharts. In C. B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer, 1984.
4. Gérard Berry. The constructive semantics of pure Esterel. Draft book, 1999.
5. Gérard Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
6. Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
7. Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, June 1999.
8. Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.
9. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications: International Conference Proceedings*, volume 735 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, June 1993. Springer.
10. Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, July 2010.
11. Randal E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(4):634–649, July 1987.
12. Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer, 1995.
13. Janusz A. Brzozowski and Michael Yoeli. On a ternary model of gate networks. *IEEE Transactions on Computers*, C–28(3):178–184, March 1979.
14. Joseph Tobin Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Proceedings of the Asilomar Conference on Signals, Systems & Computers*, pages 508–513, Pacific Grove, California, October 1994.
15. Adam Cataldo, Edward Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. Discrete-event systems: Generalizing metric spaces and fixed point semantics. Technical Report UCB/ERL M05/12, University of California, Berkeley, April 2005.
16. Adam Cataldo, Edward Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems*, July 10-12 2006.
17. Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, April 1932.

18. Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.
19. Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.
20. Patrick Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Rapport de Recherche IMAG–RR–88, Laboratoire d'Informatique, Université Scientifique et Médicale de Grenoble, Grenoble, France, September 1977.
21. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
22. Stephen A. Edwards. Concurrency and communication: Lessons from the SHIM project. In *Proceedings of the Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS)*, volume 5860 of *Lecture Notes in Computer Science*, pages 276–287, Newport Beach, California, November 2009. Springer.
23. Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, July 2003.
24. Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, August 2006.
25. Stephen A. Edwards, Richard Townsend, and Martha A. Kim. Compositional dataflow circuits. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Vienna, Austria, September 2017.
26. E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90–99, March 1965.
27. Michael J. Gordon, Robin J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.
28. Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Report on the programming language Haskell, A non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5):1–164, May 1992.
29. Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
30. Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.
31. Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, December 2002.
32. Jean-Louis Lassez, V. L. Nguyen, and E. A. Sonnenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, May 1982.
33. Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.
34. Edward A. Lee and Eleftherios Matsikoudis. The semantics of dataflow with firing. In *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, chapter 4, pages 71–94. Cambridge University Press, Cambridge, UK, 2008.
35. Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
36. Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

37. Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
38. Edward A. Lee and Pravin Varaiya. *Structure and Interpretation of Signals and Systems*. Addison-Wesley, Reading, Massachusetts, 2003.
39. Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 114–123, Austria, September 2007.
40. Sharad Malik. Analysis of cyclic combinational circuits. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 618–625, Santa Clara, California, November 1993.
41. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.
42. Michael Mendler, Thomas R. Shiple, and Gŕard Berry. Constructive Boolean circuits and the exactness of timed ternary simulation. *Journal of Formal Methods in System Design*, 40(3):283–329, June 2012.
43. D. E. Muller. Treatment of transition signals in electronic switching circuits by algebraic methods. *IRE Transactions on Electronic Computers*, EC-8(3):401, September 1959.
44. David A. Penry and David I. August. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference*, pages 926–931, Anaheim, California, June 2003.
45. Robert Pollack. Polishing up the Tait-Martin-Löf proof of the Church-Rosser theorem. In *Proceedings De Wintermöte*, Göteborg, Sweden, January 1995. Department of Computing Science, Chalmers University.
46. Dana Scott. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*, Lecture Notes in Mathematics, pages 97–136. Springer, 1972.
47. Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, September 1976.
48. Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, pages 328–333, Paris, France, March 1996.
49. Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Massachusetts, 1977.
50. Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report AIM-349, MIT AI Lab, Cambridge, Massachusetts, 1975.
51. Masako Takahashi. Parallel reductions in $\lambda$-calculus. *Journal of Symbolic Computation*, 7(2):113–123, 1989.
52. Masako Takahashi. Parallel reductions in $\lambda$-calculus. *Information and Computation*, 118(1):120–127, April 1995.
53. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.
54. Alan M. Turing. Computability and $\lambda$-definability. *The Journal of Symbolic Logic*, 2(4):153–163, December 1937.
55. Jiawang Wei. Parallel asynchronous iterations of least fixed points. *Parallel Computing*, 19(8):886–895, August 1993.
56. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, Cambridge, Massachusetts, 1993.