

The Duality of Time and Information

Vaughan R. Pratt*
Stanford University
pratt@cs.stanford.edu

June 1, 1992

Abstract

The states of a computing system bear information and change time, while its events bear time and change information. We develop a primitive algebraic model of this duality of time and information for rigid local computation, or straightline code, in the absence of choice and concurrency, where time and information are linearly ordered. This shows the *duality* of computation to be more fundamental than the *logic* of computation for which choice is disjunction and concurrency conjunction.

To accommodate flexible distributed computing systems we then bring in choice and concurrency and pass to partially ordered time and information, the formal basis for this extension being Birkhoff-Stone duality. A degree of freedom in how this is done permits a perfectly symmetric logic of computation amounting to Girard's full linear logic, which we view as the natural logic of computation when equal importance is attached to choice and concurrency.

We conclude with an assessment of the prospects for extending the duality to other organizations of time and information besides partial orders in order to accommodate real time, nonmonotonic logic, and automata that can forget, and speculate on the philosophical significance of the duality.

1 Introduction

The behavior of an automaton is to alternately wait in a *state* and perform a transition or *event*. We may think of the state as bearing information representing the “knowledge” of the automaton when in that state, and the event as modifying that information. At the same time we may think of the event as taking place at a moment in time, and the state as modifying or whiling away time.

*This work was supported by the National Science Foundation under grant number CCR-8814921 and a gift from Mitsubishi.

Thus states *bear* information and *change* time, while events *bear* time and *change* information.

We often speak of events as “time-stamped;” we might similarly speak of states as “information-stamped.”

One way of viewing this is to think of computational behavior as motion or flow in a space whose two dimensions are time and information, which we will conventionally plot horizontally and vertically respectively. Horizontal motion denotes a state, a quiescent period in which time passes while information is held constant. Vertical motion denotes an event, a transient phenomenon in which information changes but time does not. This accounts at least for rigid local behavior or nonbranching sequential computation; to extend it to flexible distributed behavior we might extend the linear geometry of each of time and space to spaces that can branch and increase in dimension, and to permit computation to be a broad flow like a river as opposed to motion of just a point. In this paper we confine attention to motion upwards (monotonically increasing information) and to the right (monotonically increasing time), leaving open the proper algebraic treatment of downward motion (forgetful automata and nonmonotonic logic), and raising as an interesting philosophical question what meaning might be attached to motion to the left, a feature of both Feynman diagrams and time machines.

Conventional automata are asymmetric with respect to time and information: states are vertices (points, 0-cells) while events are edges (line segments, 1-cells). Petri nets [Pet62, Rei85] on the other hand are symmetric: both states and events are vertices of a bipartite graph. The states, called places, are on one side, the transitions are on the other. The edges of a Petri net denote neither states nor events but rather connections between places and transitions. Edges from places to transitions specify static preconditions for a transition to *fire*, those from transitions to places specify the effect of the firing of a transition on the places. A Petri net may “be in” multiple places at the same time, represented by tokens placed on those places it is “in,” the “token game.” A conventional automaton can then be viewed as the special case of a Petri net whose every transition has both indegree and outdegree one, and corresponds to computation without concurrency. The dual notion in which every place has indegree and outdegree one is called an *occurrence net* or *causal net*, and corresponds to conflict-free computation, the notion of a deterministic computation or particular run of a net.

The token game can be formalized via a notion of global state as the multiset of marked places (two or more tokens may occupy the same place simultaneously). This however raises the “true concurrency” question of whether simultaneity of remote events is well-defined. One is therefore interested in models that are equally formal but manage somehow to sidestep this question. The occurrence subnets of an “unfolded” net, viewed as one run of the net, provide such a notion.

The algebra of Petri nets, under what we may view as their monotone com-

binators including (asynchronous) concurrency and choice, has been elegantly worked out, most notably by Winskel [Win86]. One might compare this “programmer’s” algebra to the algebra of regular expressions [Kle56], whose operations are all monotone and analogous to program connectives.

Elsewhere [Pra90] we described a conservative extension of regular expressions to what we called “action logic,” by adding two implications, *had-then* and *if-ever*. This made it into a logic by introducing nonmonotonicity, from which two negations, *never-before* and *never-again*, are derivable. The chief improvements over regular expressions are that the resulting equational theory is finitely axiomatizable (due to the nonmonotonicity permitting the expression of induction), and that it uniquely determines star in terms of union and concatenation, which the equational theory of regular expressions fails to do even in finite models (a four-element counterexample suffices).

Here we analogously extend the monotone algebra of Petri nets to a full logic, by adding implication to introduce nonmonotonicity as with action logic, this time with only one implication, and deriving a negation, namely the duality of schedules and automata, also describable as the duality of time and information. This extension is not strictly a conservative extension, the monotone operations being only loosely related to those of the extant algebras of Petri nets. The resulting logic while resembling action logic in some respects is much closer to Girard’s full linear logic, in particular satisfying De Morgan’s laws and thus having a Boolean-like symmetry of *and* and *or* absent from action logic.

In our system the Petri net formalism of a single bipartite graph is replaced by a dual pair of graphs, the schedule and its dual automaton. A *schedule* is a set of events distributed in time (temporal space). An *automaton* is a set of states distributed in information space. This reformulation is foreshadowed in Nielsen, Plotkin, and Winskel [NPW81], where Birkhoff-Stone duality makes its first albeit cryptic appearance in the theory of concurrent computation. The algebraic advantages of this passage from one bipartite graph to two dual graphs might loosely be compared to those of the passage from Aristotelian syllogistic to Boolean logic.

In practice events are distributed in space as well as time. Other than remarking that space seems to us to belong to the temporal side of the time-information duality, we shall make no attempt in this paper to incorporate space into our picture.

The information side of this duality ties in very satisfactorily with (Scott) domain theory, which emphasizes elements partially ordered by information content. The viewpoint of this paper adds the further interpretation that those elements are states, as opposed say to recursively defined functions, the motivating interpretation of domain theory.

It is possible, though not necessary, to make the duality of time and information perfectly symmetric, via our notion of event and state spaces. A considerable portion of domain theory (though not that involving stable functions) then can be reflected via this duality to apply not only to “information

systems” but equally well to “temporal systems.”

2 Rigid Local Computation and Chains

Straightline code contains neither branches nor concurrency but merely specifies a fixed sequence of operations. We think of a nonbranching program as rigid or inflexible, and one without concurrency as local or sequential.

Yet even in this computationally sterile setting one can not only find the duality of time and information but treat it algebraically. This simple setting has the virtue of revealing some of the basic principles of the duality without the distraction of such issues as choice and concurrency, respectively the disjunction and conjunction of computation. In this respect then the *duality* of time and information is more fundamental than its *logic*.

We now study certain categories of chains and their maps. The chains will be used to formalize both schedules and automata, with their elements corresponding respectively to events and states, and with the edges between consecutive elements corresponding respectively to states and events. Our goal here is to work out in detail the mathematics of this cross-connection, both as interesting mathematics in its own right and as a special case of a similar but richer cross-connection in the case of flexible distributed computation.

This paper is written for an audience interested in approaches to the formalization of concurrency, and assumes relatively little algebraic sophistication or familiarity with categories.. That we bring in category theory at all reflects the nature of duality as categorical rather than set-theoretic. The categories of finite chains that we begin with are simple enough that any mathematically mature reader should be able to follow every step and acquire the necessary category theory along the way.

A *chain* is a linearly ordered set $C = (X, \leq)$. For simplicity of exposition we shall restrict ourselves in this section to finite chains. The popularity of the term “fencepost error” suggests that we visualize a chain as made up of posts (the elements) and fences (consecutive pairs of elements).

A *map* $f : C \rightarrow D$ of chains is a monotone function, one such that $x \leq y$ implies $f(x) \leq f(y)$. We denote by **Fchn** the category¹ of all finite chains and their maps.

A variant on **Fchn** is **Fchn0**, the category of finite chains with bottom. The objects are those objects of **Fchn** that have a least element 0, namely all but the empty chain, and its maps are those maps of **Fchn** that preserve that least element, that is, $f(0) = 0$. The dual of this is **Fchn1**, the category

¹A *category* is a reflexive multigraph, that is, a graph permitting multiple edges or *morphisms* from one vertex or *object* to another, including a distinguished self-loop at each vertex, along with an associative composition law for which the self-loops are identities. Any collection of sets and functions between them closed under ordinary function composition and containing the identity function for every set automatically forms a category. The reader should verify that our purported examples of categories are indeed so closed.

of chains with top 1 and top-preserving maps, $f(1) = 1$. The intersection of these categories, **Fchn01**, consists of chains with top and bottom, with maps preserving both.

Let \mathbf{m} denote the m -element chain $(\{0, 1, \dots, m-1\}, \leq)$ standardly ordered by \leq . Of particular importance is $\mathbf{2}$, the two-element chain. For any chain C define the chain $\mathbf{2}^C$ to consist of all maps $f : C \rightarrow \mathbf{2}$, ordered pointwise, that is, $f \leq g$ just when $f(x) \leq g(x)$ for all x in C . It is readily seen that the maps $f : C \rightarrow \mathbf{2}$ are in bijective correspondence with the fences of C : each f corresponds to the fence whose posts are the greatest element mapped to 0 and the least element mapped to 1.

There are two issues here. First, the existence of the constantly 0 function $K0$ and its dual $K1$ demonstrates the need for two additional fences not meeting our description as consecutive pairs of posts, namely the fences lying outside the whole chain at either end. In order for this bijection to exist **Fchn** must admit both outlying fences, **Fchn0** must rule out the lower fence ($K1$ does not exist), dually **Fchn1** the upper fence ($K0$ is out), and **Fchn01** must rule out both.

Second, there is the question of the order of the fences (the alert reader will have noted the paradox of the largest function $K1$ corresponding to the bottom fence and vice versa). The larger functions are those with more 1's, but those correspond to the fences closer to the bottom. Hence although it is indeed a chain of fences of C , *the pointwise order of $\mathbf{2}^C$ is the reverse of the order in C of its posts.*

Let C^\smile denote the order dual or *converse* of C , C turned upside down. Then either $\mathbf{2}^{C^\smile}$ or $(\mathbf{2}^C)^\smile$ denotes the fences of C in their natural order as defined by the order of their posts in C .

We now have three operations, $\mathbf{2}^C$, C^\smile , and their composition $\mathbf{2}^{C^\smile}$. We call these respectively the *dual*, *converse*, and *complement* of C , and denote them respectively C^\perp , C^\smile , and C^- .

Our next goal is to show that all three operations are involutions (self-inverses), and commute with each other. It is clear that converse is an involution, and we have noted that it commutes with dual. The remaining commutativities then follow by expanding the definition of complement. We now show that dual is also an involution, from which it immediately follows that complement is an involution.

One thing that dualizing does is to turn fences into posts. The posts of C then can be viewed as turning into fences of C^\perp . Although the transformation takes place in one step we could imagine it happening continuously, with the fences of C shrinking to become posts of C^\perp , and the posts of C then stretching to fill in the resulting spaces, in parallel with rotating the whole chain so as to reverse the order. Complement also has this effect, but without the side effect of order reversal.

Given a chain C of **Fchn**, if we take C^\perp to also belong to **Fchn** then C , C^\perp , $C^{\perp\perp}$, and so on will be progressively longer chains. But if we regard C^\perp

as belonging to **Fchn01**, with bottom $K0$ (the constantly 0 function) and top $K1$, then although C^\perp will be one larger than C , $C^{\perp\perp}$ will shrink back down to the size of C again, and hence be isomorphic to C . If in addition, for any chain D in **Fchn01** we regard D^\perp as belonging to **Fchn**, if C is in **Fchn** then so is $C^{\perp\perp}$. (We can rationalize this choice of category for D^\perp with the observation that D^\perp contains neither $K0$ nor $K1$ and so is a chain with neither a significant bottom nor top.) Moreover if D is in **Fchn01** so is $D^{\perp\perp}$. That is, dual switches back and forth between the two categories, and double dual is an isomorphism of **Fchn** and also of **Fchn01**. (Double dual of course reverses order twice.)

We thus have that converse is an isomorphism of **Fchn** with itself, as well as of **Fchn01** with itself, while dual is an isomorphism between **Fchn** and **Fchn01**. Converse is a true involution (composing it with itself yields the identity on **Fchn**, and similarly with **Fchn01**), while dual is an involution up to isomorphism, that is, $C^{\perp\perp}$ is isomorphic to C though not equal to it. We could turn isomorphism into equality by cutting **Fchn** and **Fchn01** down to what is called their *skeletons*, an arbitrarily chosen full subcategory consisting of one representative of each class of isomorphic objects, but the arbitrariness should serve to warn against such radical surgery.

We chose the terms “complement” and “converse” by analogy with the calculus of binary relations. Converse of course has its standard relational meaning with respect to the binary relation \leq defining a chain. The converse R^\smile of the Boolean complement R^- of a binary relation R behaves analogously to dual, when **2** is replaced by the complement $0'$ of the identity relation $1'$, and exponentiation R^S is taken to mean the left residuation operation $R/S = (R^-; S^-)^-$ where $R; S$ is composition of binary relations (the right residual $S \setminus R$ will do just as well). Although no separate term for R^\smile has emerged in the relation calculus literature, as we have pointed out elsewhere [Pra92b] the significance of this operation as a form of negation was recognized by C.S. Peirce as long ago as 1882, in a Johns Hopkins circular *Remarks on [B.I. Gilman’s “On Propositions and the Syllogism”]* [Klo86, p.345]. The same relationships obtain between relational converse, complement, and their composition thought of as dual, as with the operations of those names for chains. The two points of contact between relations and chains are via converse and dual, and the notion of complement for chains is then taken to be a derived notion by analogy with relations, namely as the composition of the other two operations. But whereas for relations complement is a Boolean operation, for chains it has no special Boolean character.

Now let us turn attention to **Fchn0** and **Fchn1**. Here dual performs the same exchange of posts and fences, but without growth or shrinkage of chains. For C in **Fchn0** the question arises as to whether C^\perp should be considered as belonging to **Fchn0** or **Fchn1**. Following our earlier reasoning about constant functions, we observe that C^\perp contains $K0$ but not $K1$, whence it belongs to **Fchn0**, $K0$ being the bottom of C^\perp . (Admittedly it is not clear why constancy is the appropriate criterion for this choice. However in the case of flexible distributed computation, where the appropriate generalizations of **Fchn0** and

Fchn1 are respectively state spaces and event spaces, this is the only possible choice.)

Converse on the other hand is a map from **Fchn0** to **Fchn1**, and its inverse (which we shall also call converse) going from **Fchn1** to **Fchn0**. Hence complement also runs between **Fchn1** and **Fchn0**.

So far we have only described the behavior of converse, dual, and complement on the objects of the categories. We shall now describe their behavior on the maps of the categories as well, making them into true functors between categories. In doing so we shall be obliged to distinguish between functors that preserve map direction and those that reverse it, called respectively covariant and contravariant functors. We shall follow the usual practice of dispensing altogether with the notion of a contravariant functor $F : A \rightarrow B$ from category A to category B by describing it instead as the covariant functor $F : A \rightarrow B^{\text{op}}$, where B^{op} , the *opposite* of B , denotes B with its maps reversed.

For more insight into the significance of reversing maps, let us consider the maps of our four categories in more detail. In particular let us count the number of maps from a given chain \mathbf{m} to \mathbf{n} . We call the set of such maps the *homset* from \mathbf{m} to \mathbf{n} , notated $\text{Hom}(\mathbf{m}, \mathbf{n})$ or $A(\mathbf{m}, \mathbf{n})$ if we wish to specify the category A to which the homset belongs.

We can easily enumerate such a homset using the following notation for a map $f : C \rightarrow D$. For each element d of D in order, list in order the elements of $f^{-1}(d)$ followed by d itself, creating a list of lists that we then “flatten” to a single list. Since f is monotone, the resulting list notating this function will be some merge of the elements of C with those of D . The last element of D must be the last element of this merge and hence is redundant, so we modify this notation to omit the last element of D . In **Fchn**, every merge of the elements of C with all but the last element of D arises in this way. There are $\binom{m+n}{m}$ merges of two lists having respectively m and n elements. Hence the number of monotone functions from the chain \mathbf{m} to the chain \mathbf{n} is $\binom{m+n-1}{m}$ (since we omitted the last element of \mathbf{n}).

As an example take the three maps from $\mathbf{2}$ to itself, namely $K0$, I (the identity), and $K1$. Write the elements of $\mathbf{2}$ as $a < b$ when $\mathbf{2}$ is the domain of the maps and as $0 < 1$ when the codomain (target). Then these three maps notated in full are respectively $ab01$, $a0b1$, and $0ab1$. When modified to omit the last element of the codomain these shorten to $ab0$, $a0b$, and $0ab$. These are all the possible merges of ab with 0 .

In **Fchn0** the first element of the domain must occur first in our notation, whence we may omit it as well, leaving only $\binom{m+n-2}{m-1}$ maps from \mathbf{m} to \mathbf{n} . But this of course equals $\binom{n+m-2}{n-1}$, whence in **Fchn0** the number of maps from \mathbf{m} to \mathbf{n} equals the number from \mathbf{n} to \mathbf{m} . Put differently, **Fchn0** has the same number of maps from \mathbf{m} to \mathbf{n} as **Fchn0**^{op}. Hence as graphs (i.e. ignoring the composition law making a graph into a category), **Fchn0**^{op} is isomorphic to **Fchn0**. (Recall that we are allowing a graph to have a set of edges from one

vertex to another.)

When A is isomorphic to B^{op} we say that A is *dual* to B . Thus we have shown that **Fchn0** is *self-dual* as a graph.

This proof is not constructive in that it does not specify any particular bijection of the set of edges from \mathbf{m} to \mathbf{n} with the equinumerous set from \mathbf{n} to \mathbf{m} . In the case of **Fchn0**, note that the first element of m and the last element of n is omitted, and dualizing reverses order. Thus an obvious choice of f^\perp is just the function represented by writing our notation for f in reverse order. For example among the six functions from **3** to itself is $0bc1$ (mapping a to 0 and b and c to 1), whose dual is therefore $1cb0$ mapping 2 and 1 to c and 0 to a . (We adopt the convention, at least for **Fchn0**, of naming a function to **2** by the largest element it sends to 0. With this convention in **Fchn0** the dual of the chain 012 is written simply 210.)

In fact **Fchn0** is self-dual as a reflexive graph, noting that every chain has at least one map to itself, namely the identity map, and that the above bijection pairs up identities.

A similar duality holds between **Fchn** and **Fchn01**. The reader may verify that in **Fchn01** there are only $\binom{m+n-3}{m-2}$ maps from \mathbf{m} to \mathbf{n} . But the dual of \mathbf{m} in **Fchn** grows to $\mathbf{m} + 1$ in **Fchn01**, so the set of $\binom{m+n-1}{m}$ maps from \mathbf{m} to \mathbf{n} in **Fchn** should be compared with the set of $\binom{(n+1)+(m+1)-3}{n-1}$ maps from $\mathbf{n} + 1$ to $\mathbf{m} + 1$, and indeed these quantities are equal.

We now show that **Fchn0** is self-dual not only as a reflexive graph but as a category, by extending C^\perp to a functor and then showing that it is still an isomorphism, but now between **Fchn0** and **Fchn0**^{op} rather than from **Fchn0** to itself (i.e. the object part of C^\perp did not tell the whole story).

Given $f : C \rightarrow D$ in **Fchn0**, define 2^f , or f^\perp , to be the function which, given a function $g : D \rightarrow \mathbf{2}$, i.e. an element of D^\perp , yields the function $g \circ f : C \rightarrow \mathbf{2}$ (an element of C^\perp), monotone since composition preserves monotonicity. This then defines the dual of $f : C \rightarrow D$ to be a function $f^\perp : D^\perp \rightarrow C^\perp$. Since C^\perp and D^\perp are objects of **Fchn0** and f^\perp is a monotone function between them, this makes f^\perp a map of **Fchn0**, but running contravariantly to f .

We now have two competing very reasonable ways to create a bijection between **Fchn0**(\mathbf{m}, \mathbf{n}) and **Fchn0**(\mathbf{n}, \mathbf{m}), notation reversal and dualization. Fortunately they are the same (exercise). The advantage of the latter description of this bijection is that it makes it easy to show that the bijection is functorial (is a homomorphism with respect to composition), as follows.

$$\begin{aligned}
(g \circ f)^\perp(h) &= h \circ (g \circ f) \\
&= (h \circ g) \circ f \\
&= f^\perp(h \circ g) \\
&= f^\perp(g^\perp(h)) \\
&= (f^\perp \circ g^\perp)(h)
\end{aligned}$$

(The interchange of f and g is a side effect of f^\perp being a duality as opposed to an isomorphism.)

We now similarly extend converse to a functor, in the process showing that **Fchn0** is isomorphic to **Fchn1** as a category. Define the converse of $f : C \rightarrow D$, namely $f^\smile : C^\smile \rightarrow D^\smile$, to be the same function as f on the underlying sets of C and D . Since the order of both domain and codomain have been reversed, f^\smile remains monotone. And this description of converse makes it clear that $(g \circ f)^\smile = g^\smile \circ f^\smile$, and that the converse of the identity map is still the identity map. Hence converse is functorial.

Since complement is the composition of converse and dual, it follows that complement is also a functor.

We then have the following two diagrams showing all the above isomorphisms as functors between the indicated categories.

$$\begin{array}{ccc}
 \mathbf{Fchn} & \overset{\smile}{\leftrightarrow} & \mathbf{Fchn} \\
 \perp \updownarrow & & \perp \updownarrow \\
 \mathbf{Fchn01}^{\text{op}} & \overset{\smile}{\leftrightarrow} & \mathbf{Fchn01}^{\text{op}}
 \end{array}
 \qquad
 \begin{array}{ccc}
 \mathbf{Fchn0} & \overset{\smile}{\leftrightarrow} & \mathbf{Fchn1} \\
 \perp \updownarrow & & \perp \updownarrow \\
 \mathbf{Fchn0}^{\text{op}} & \overset{\smile}{\leftrightarrow} & \mathbf{Fchn1}^{\text{op}}
 \end{array}$$

In each diagram the complement functor, as the composition of converse with dual, or vice versa, runs along both diagonals. Technically speaking all arrows with distinct domain or codomain should be considered distinct, so in the case of the right hand diagram we really have a total of twelve functors consisting of four distinct clones of each of converse, dual, and complement. The left diagram has only six distinct functors since each appears twice.

Since each of **Fchn0** and **Fchn1** are self-dual, unlike either of **Fchn** and **Fchn01**, all four of **Fchn0**, **Fchn1**, **Fchn0**^{op}, and **Fchn1**^{op} appear in the one diagram, and hence all four are isomorphic to one another via the indicated isomorphisms. In the case of **Fchn** and **Fchn01** however, the only isomorphisms we have are between **Fchn** and **Fchn01**^{op}; there is an analogous but separate set of isomorphisms between **Fchn**^{op} and **Fchn01** which we did not bother to diagram. So counting only up to isomorphism, we have only three nonisomorphic categories of chains, **Fchn**, **Fchn01**, and **Fchn0**.

None of these categories have products or sums (coproducts), except for the empty sum or initial object (the object with exactly one map to every object including itself) and the empty product or final object (dually the object with exactly one map from every object including itself). In the case of **Fchn** the empty chain is initial and the unit chain is final. For **Fchn01** the two element chain is initial (both 0 and 1 must be sent to 0 and 1 in any map) and the one element chain is final (while any map to the unit chain is uniquely determined there are no maps from the one element chain to any chain of **Fchn01** except itself); in **Fchn01**^{op} these are reversed. The isomorphism between **Fchn** and **Fchn01**^{op} must therefore on these structural grounds pair the empty schedule with the unit automaton and the unit schedule with the two-element automaton (but we knew this already anyway).

In **Fchn0** and **Fchn1** the unit chain can be seen to be both initial and final, that is, a *zero* object.

3 Computational Interpretation of Chains

In any of the dualities we have seen, we want one chain to play the role of schedule and its complement the role of the corresponding automaton. There are four ways to do this depending on which of our four categories of chains we take as supplying the schedules.

When we choose **Fchn** for the schedules we are saying that a computation is a schedule S consisting of a finite sequence of 0 or more events. The corresponding complementary automaton S^- then consists of one or more states indicating how many events have been performed thus far. S^- has both a first and last state, which coincide just for the case when S is empty. In the first state nothing has been done, while in the last state everything is finished.

Now consider the choice of **Fchn01** for the schedules. What significance are we to attach to the initial and final events? Well, look at the complementary automaton to see what it permits. The first state records that the initial event has already been done, while the last state records that the final event still has not been done. We infer that the initial event is one that has always been done (was never not done), while the final event will never be done, the mathematical version of “over my dead body.”

It is quite reasonable to have these “preordained events,” since they can be used as single placeholders for the collection of all events of a schedule that we wish to consider as earlier than we care to think about, and dually those events we wish to imagine as happening later than we are interested in. This can be quite useful when using maps to massage or edit schedules. Each schedule consists of the pre-event, the active events, and the post-event. We think of a map as describing an editing operation in which the domain of the map is the schedule to be edited and the codomain the result of editing it. (Don’t think of this as a function describing a generic editing operation applicable to many schedules, the map includes the particular schedule the operation was applied to. This is in general a good way to think about the role of maps when thinking of an object as an individual datum as opposed to a type of data.) Maps can add new active events, delete active events by moving them earlier to merge in with the pre-event or later to merge with the post-event, and combine active events. They can also combine the pre-event and post-event, but this yields inconsistency and prevents any further editing.

Now consider **Fchn1** for the schedules. Combining the reasoning for the two previous cases we infer that each schedule S has an impossible final event while its complementary automaton S^- , which must be in **Fchn0^{op}**, has an initial state in which nothing has yet been done, but no final state in which everything has been done. If we reverse this by taking **Fchn0** for the schedules then S has

an always-done initial event while S^- has a final state in which everything has been done, but no initial state in which nothing has been done.

One can imagine uses for all four of these choices. If one's primary programming language is schedules rather than automata then **Fchn** is a natural choice in that always-done and never-done events might seem like redundant bells and whistles. As much could be said for automata, forcing the choice of **Fchn01**^{op} for schedules. But if one wants schedules and automata to be perfectly symmetric then one assigns **Fchn0** and **Fchn1**^{op} to them, one way round or the other.

4 Flexible Distributed Computation and Birkhoff-Stone Duality

For conventional automata the passage from linear to nonlinear computation is associated with the introduction only of branching as choice. For the notion of automaton that we shall treat in this paper the passage from linear computation as chains to nonlinear computation as posets is associated with the introduction of both branching and concurrency.

On the mathematical side, our categories of chains are not closed under three natural operations that we would like to bring in: sum, product, and exponentiation (though they are closed under equalizers and coequalizers). It is natural to think of the sum of two schedules, and dually the product of two automata, as their concurrent or conjunctive composition. The sum of automata (and hence by duality, but less obviously, the product of schedules) should represent their alternative or disjunctive composition. Exponentiation A^B is also useful, as the system resulting from A observing B . **Fchn** is not closed under the sum of two nonempty chains, the product of two chains with two or more elements, nor under C^D for C with at least three elements and D at least two. With small variations in these parameters the other categories of chains are similarly not closed.

The problem is the assumption of linearity of order, which so far has helped us by keeping the model very simple. Merely dropping it, thereby passing to partial orders, yields closure under sum and product, and in suitable analogues of **Fchn0** and **Fchn1** also exponentiation, which seems to thrive on symmetry.

Fchn most naturally turns into the category **Fpos** of finite posets (partially ordered sets), while its dual **Fchn01** turns into the category **FDL** of finite distributive lattices with top and bottom. The operations of converse, dual, and complement that we described for chains carries over without any essential changes, yielding diagrams of isomorphic categories exactly analogous to those for chains. Converse A^\smile continues to be the result of reversing order, whether of posets or distributive lattices, and is obviously an isomorphism of **Fpos** with itself and of **FDL** with itself. Dual continues to be defined as before: if P is

a finite poset, 2^P is a finite distributive lattice, and vice versa, making **Fpos** dual to **FDL** (isomorphic to **FDL**^{op}) just as for **Fchn** and **Fchn01**, shown by Birkhoff in 1933 [Bir33]. Stone [Sto37] a little later found one extremal extension of this duality to infinite objects, much later characterized nicely by Priestley [Pri70] as the duality of partially ordered Stone spaces and distributive lattices; the other extremal extension is between posets and *profinite* distributive lattices, for whose definition, history, and many further extensions see Johnstone [Joh82, Ch.VII].

Complement continues to be 2^{A^\vee} . All three of converse, dual, and complement continue to commute with each other and be involutions. And the same arguments showing that they are functorial continue to apply.

Fpos and **FDL** are each closed under finite sum (juxtaposition in the case of **Fpos**) and finite product (cartesian product in both cases), and the passage to infinite objects is then accompanied by an extension of sum and product to infinite arities referred to as closure under arbitrary sums and products. (An infinite sum of nonempty finite posets is necessarily infinite, and likewise for products of posets with at least two elements.) With regard to closure under exponentiation the situation is as for **Fchn** and **Fchn01**: like **Fchn** we may consider **Fpos** to be closed under exponentiation (since the set of monotone functions from poset P to poset Q forms a poset under pointwise order), but prefer not to since as we have seen we wish to view 2^P as a distributive lattice, just as we viewed 2^C as in **Fchn01** even though it was eligible for **Fchn**. And since any poset can arise as 2^L for some L in **DL**, **DL** is certainly not closed under exponentiation.

We now consider corresponding generalizations for **Fchn0** and **Fchn1**. It will turn out that in this symmetric situation infinite objects are handled with less fuss than in any previous situation, allowing us to dispense with the pedagogy of starting out with finite objects. And as we remarked earlier the symmetry also greatly helps exponentiation.

The most straightforward generalization of **Fchn0**, chains with the empty join (bottom), is to **CSLat**, posets with arbitrary joins, called join-complete semilattices. Their order dual, posets with arbitrary meets, called meet-complete semilattices, is isomorphic to itself, to its opposite, and to **CSLat**, whence if you are of the school of thought that routinely takes skeletal categories, i.e. recognizes only two groups of order four and one set of cardinality four, you would not give it a separate abbreviation (nor would you distinguish the categories **Fchn0** and **Fchn1**). For convenience in distinguishing the three isomorphisms of converse, dual, and complement we shall distinguish these anyway as **CSLat**_∨ and **CSLat**_∧. With regard to the three isomorphisms these are the exact analogues of **Fchn0** and **Fchn1** respectively, the corresponding diagram being as follows.

$$\begin{array}{ccc}
 \mathbf{CSLat}_\vee & \overset{\sim}{\leftrightarrow} & \mathbf{CSLat}_\wedge \\
 \perp \updownarrow & & \perp \updownarrow \\
 \mathbf{CSLat}_\vee^{\text{op}} & \overset{\sim}{\leftrightarrow} & \mathbf{CSLat}_\wedge^{\text{op}}
 \end{array}$$

These being isomorphic, it suffices to consider closure properties of **CSLat**. **CSLat** can be readily shown to be closed under arbitrary (including empty and infinite) sums and products (indeed under all limits and colimits), as well as under exponentiation.

However it turns out that sum and product are the same operation in **CSLat**. A logic using these connectives for disjunction and conjunction would be undesirably degenerate: what use would *or* be when it meant the same as *and*?

There is a small variation on this generalization that has exactly the right effect. Instead of generalizing **Fchn0** to finite join semilattices, use posets that have bottom (as with **Fchn0**), but whose nonempty subsets have meets instead of joins (and with maps preserving bottom and nonempty meets). We have elsewhere [Pra91a, Pra92a] called such a structure a *state space*, forming the category **St**. The dual notion is a poset with top and all nonempty joins, called an *event space*, forming the category **Ev**. The isomorphisms are thus:

$$\begin{array}{ccc} \mathbf{St} & \overset{\sim}{\leftrightarrow} & \mathbf{Ev} \\ \perp \uparrow & & \perp \uparrow \\ \mathbf{St}^{\text{op}} & \overset{\sim}{\leftrightarrow} & \mathbf{Ev}^{\text{op}} \end{array}$$

All the properties we have enumerated thus far for complete semilattices also obtain for event and state spaces, except for the degeneracy of *or* and *and*. Converse reverses the order, while the dual of A is still 2^A and complement is their composition.

There is a simple alternative description of complement of an event space. Simply remove ∞ from the top and install q_0 at the bottom, leaving the other elements unchanged. Complement of chains can be seen to be a special case of this. The patient reader can confirm this by direct calculation with a few examples, for a proof see [Pra91a].

But this suggests that all the active (non- ∞) events of an event space actually *become* the active (non- q_0) states of a state space. With the chains we were imagining that the fences between the elements (as posts) of a schedule were states, and that complementation turned those fences into posts and vice versa. In this new view we are leaving the whole of the active structure fixed and adjusting only the top and bottom. This new view appears to represent a slight phase shift. We do not have a good explanation of this shift, although it seems to us that there should be some way to explain the duality in the same fence-post terms that worked well for chains.

Now has this very small variation on **CSLat** broken the degeneracy sufficiently to be useful? Indeed it has, as we shall now argue briefly, more detailed discussion appears elsewhere [Pra91a, Pra92a].

In an event space, we take the final event, denoted ∞ , to be the never-done event, as in **Fchn1**, and we take the join of a nonempty set Y of events, denoted $\bigvee Y$, to be the event expressing the completion of all events in the set. It is possible that $\bigvee Y = \infty$, in which case we say that Y is in *conflict*: it is not

permitted for all events of Y to finish. This gives us a way of expressing conflict between events that a naive notion of schedule as a poset of events does not offer.



Dually in a state space, we take the initial state, denoted q_0 , to be the state of ignorance (“original sin”), and we take the meet of a nonempty set Y of states, denoted $\bigwedge Y$, to be the last state at which every state of Y remains a possible future state. It is possible that $\bigwedge Y = q_0$, in which case we say that Y is in *dilemma*: knowing nothing, the automaton is nevertheless obliged to choose a proper subset of Y .

These notions of conflict and dilemma are internal to event and state spaces. External are the notions of sum, product, and exponential used to form larger spaces from smaller.

We can gain some insight into how these operations work by applying them to two-element event and state spaces, each of which represents one “active” event or state and one “dummy,” either the final event or the initial state. Because they contain only one active event or state we think of them as the unit event space and unit state space respectively.

The sum and product of two unit event spaces are given respectively by

$$\begin{array}{c} \bullet \\ | \\ \bullet \end{array} + \begin{array}{c} \bullet \\ | \\ \bullet \end{array} = \begin{array}{c} \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \end{array} \quad \text{and} \quad \begin{array}{c} \bullet \\ | \\ \bullet \end{array} \times \begin{array}{c} \bullet \\ | \\ \bullet \end{array} = \begin{array}{c} \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \end{array},$$


where time is assumed to flow from bottom to top. Sum represents asynchronous concurrent composition, with the leaves of  representing the two concurrent events, their join representing their concurrent execution, and ∞ representing the unperformed event. Product represents choice, with the basic events being the two side events of , in conflict because their join is ∞ , and with the bottom event denoting the information needed to make the choice.


The same concepts appear in complementary form on the automaton side. In lieu of sum and product we have the product and sum of two unit state spaces given respectively by

$$\begin{array}{c} \bullet \\ | \\ \bullet \end{array} \times \begin{array}{c} \bullet \\ | \\ \bullet \end{array} = \begin{array}{c} \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \end{array} \quad \text{and} \quad \begin{array}{c} \bullet \\ | \\ \bullet \end{array} + \begin{array}{c} \bullet \\ | \\ \bullet \end{array} = \begin{array}{c} \bullet \quad \bullet \\ \backslash \quad / \\ \bullet \end{array}.$$

The product automaton can be thought of as accepting $ab+ba$, the “interleaving”²

²The representation of concurrent composition as interleaving may seem like a violation of true concurrency. The solution we have proposed elsewhere [Pra91b] to this apparent mismatch in the duality is to regard the product automaton as a 2D surface rather than a hollow square along the lines of [Pap86] and [Shi85].

of the basic events a and b , with the northwest axis corresponding to the a transition and the northeast the b . The sum denotes choice; just as  contains

no conflict, so does  contain no dilemma: it first performs a transition interpretable as the gathering of information for the choice, and then chooses one of the two transitions at the branch.

5 Linear Logic

We may find linear logic [Gir87] in event spaces in a natural way as follows. We start with two primitive operations: product $A \times B$, called *with* by Girard, defined as cartesian product of event spaces, and exponentiation A^B , called linear implication and written $A \multimap B$, defined as the event space of all event space maps from A to B .

We also have two primitive constants 1 and 2 denoting the event spaces of those cardinalities. The two so-called additive constants of linear logic, the respective units of the additive connectives, are both 1 , while the two multiplicative constants are both 2 . These are the only glaring degeneracies in this model of linear logic.

We then derive dual (called *perp* in linear logic) as $A^\perp = A \multimap 2$. Next we obtain sum $A + B$ as the De Morgan dual of product, $A + B = (A^\perp \times B^\perp)^\perp$. Sum and product are linear logic's "additive" connectives.

We next define tensor product $A \otimes B$ via $A \otimes B = (A \multimap B^\perp)^\perp$, and the tensor sum $A \oplus B$ (Girard notates this with an inverted ampersand) as the De Morgan dual of tensor product, $A \oplus B = (A^\perp \otimes B^\perp)^\perp$. These are the multiplicative connectives. The meaning of $A \otimes B$ is the flow of A through B , a symmetric relationship.

We now define a third primitive operation: $!A$ is the free event space on (generated by) the underlying poset of A , see [Pra91a, Pra92a] for details. Lastly we derive $?A$ as its dual, $?A = (!A^\perp)^\perp$, and an additive or intuitionistic implication $A \Rightarrow B$ defined by $A \Rightarrow B = !A \multimap B$.

To remove these last degeneracies, take a larger model containing both event and state spaces mingled. Think of this as the product of \mathbf{Ev} with the two-element category 2 with objects $0,1$ and one nonidentity map from 0 to 1 , and on which the operations of linear logic are all assigned their natural Boolean interpretation. For the event space A , interpret $(A,1)$ as an event space and $(A,0)$ as a state space. The operations are now all determined: dual for example becomes complement, while the product of an event space with a state space is a state space, which can be seen to be calculated by taking the converse of the event space to make it a state space and then multiplying by the state space. It

should be noted that the same trick when applied to **CSLat** also removes some of its degeneracies, though not the degeneracy $!A=?A$.

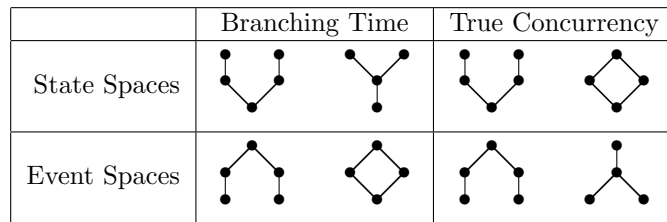
6 The Duality of Branching Time and True Concurrency

Branching time refers to the timing of decisions in semantic models: can all decisions be regarded as having been made at the beginning of time, or should the model record when (relative to other operations) a given decision was committed to? Prior to Milner’s work on CCS this timing was not recognized as an essential feature of a semantic model, and continued not to be so recognized during the eighties, witness the number of papers in temporal logic that pitted linear time against branching.

True concurrency refers to whether there is any branching other than decision branching. That is, given the outcome of all decisions, is the resulting completely deterministic computation then a linear sequence of events? Or can two events occur in no particular order, suggesting that a deterministic computation might be a partially rather than linearly ordered set of events? Certainly there exist independent events, but their relative timing could be considered an either-or proposition—the events can happen in either order, as opposed to no well-defined order—with the (admittedly noncausal) choice of their relative order being lumped together with all the other choices. True concurrency takes the position that “either order” is different from “no order.” Like branching time, true concurrency has its proponents and opponents.

We demonstrate here a connection between branching time and true concurrency that makes a much stronger connection between them than as mere coexisting imponderables of the eighties. We shall show that they are in fact dual phenomena structurally. That is, the dual of one in the sense of this paper *is* the other.

The following diagram depicts branching time on the left and true concurrency on the right. The upper row gives the state space perspective, the lower row the complementary event space account. In each of these four groups of two figures, the left figure is “before” or “bad” and the right “after” or “good.”



In the state space depiction of branching time (upper left), we give the conventional automata distinguishing $ab + ac$ from $a(b + c)$ (the initial state is at

the bottom). The corresponding event spaces below are obtained as usual by deleting the initial state and installing the final event.

In the event space depiction of true concurrency (lower right), we have on the left two sequences of two events, corresponding to the choice ab or ba . On the right is the event space expressing the truly concurrent execution of a and b . The state spaces above are obtained as before.

The striking feature of this diagram is that the branching time contrast for state spaces is the order dual of the true concurrency contrast for event spaces, and vice versa (the other cross-connection), as promised.

We have shown only the structural similarity (no labels). The difference is that whereas with branching time the two sequences are ab and ac , with true concurrency they are ab and ba . The duality of branching time and true concurrency is therefore a structural one, and a distinction emerges when labels are introduced. Labels however are beyond the scope of this paper, which has focused exclusively on the underlying structure.

7 Future Work

The “automata” we have presented here are really “unfolded” automata, at least with respect to choice and iteration. Any choice leads to disjoint sets of states, the automaton accepting $a + b$ must be implemented with three states rather than two. (We think of the two-state version as one that, having made a choice of transitions, forgets the choice by coming back to a fixed state independent of that choice.) Furthermore iteration must be unwound, no state may be visited twice in a computation.

The question then naturally arises, can this duality be extended to handle automata that forget, and/or automata containing cycles?

One direction to pursue here is Pontryagin duality in locally compact Abelian groups. Consider the group G of complex numbers on the unit circle under multiplication. It is the dualizer (in the sense that the dual of H is G^H) for Pontryagin duality. Its own dual is the group of integers under addition, while the group of reals under addition is self-dual.

Groups rather than monoids make sense for automata that can forget because the inverse gives a means of taking things back, whether information or time. And of course groups are a natural setting for cyclic behavior.

There is also a connection with nonmonotonic logic here, whose essential characteristic is its ability to take back information that has accumulated in a theory. We view the problem of duality for forgetting automata and the problem of formalizing nonmonotonic logic as at least intimately related if not in fact the same problem.

Lastly we mention real time. We have only discussed ordered temporal and information spaces, whose metric is essentially two-valued: one event either does or does not precede another, and one state either does or does not contain

less information than another. A natural extension of this notion that we have pursued elsewhere [CCMP91] is to richer measures of temporal distance between events such as causal time and real time. We have not explored the connection between these rich temporal structures with the duality of time and information, but it seems to us that such a connection should yield much additional insight into the nature of computation broadly construed to cover a wide range of such metric spaces of events and states.

8 Philosophical Significance

It seems that almost every proverb has its counterproverb. The sayings “Time is money” and “Look before you leap” express a tradeoff between response time and information gathered.

Theoretical computer science, at least at its most incestuous, i.e. not concerned for specific applications, can be broadly divided into those concerned about performance, the complexity theorists, backed by combinatorics, and those concerned about information without regard for time, once taken to be automata theory and formal languages but nowadays semantics and verification, backed by logic. This boundary of course is not hard and fast, and the advent of probabilistic computing has led to some rapprochement between these two foci. But it would seem that probabilistic computing gives up such a tiny amount of confidence in the answer that the rapprochement should be greatly improvable by giving up a much greater degree of confidence in return for yet more time.

In the silicon business, time to market is critical, but so is the correctness of large chips, calling for a similar juggling of priorities. This generalizes readily to almost any line of work: accuracy and response time are almost always both important.

A strikingly similar duality is found in quantum mechanics, where time is dual to energy, and space to momentum, to name two dualities. By setting $c = 1$ to eliminate the unit of time by equating nanoseconds to feet, the analogous complementarity of space and momentum becomes the same complementarity (but in 3D) by conferring its units, say CGS, namely cm and dyne-sec (gm-cm sec^{-1}), on time and energy respectively. By setting Planck’s constant $\hbar = 1$ the now common unit of momentum and energy becomes simply cm^{-1} . These two simplifications are frequently adopted in quantum field theory.

We could just as well take as the unit of space-time the second, a mere 3×10^{10} longer, making the unit of momentum-energy sec^{-1} or Hz. This brings quantum mechanics within striking distance of computation: we are comparing QM’s duality of time and energy with computation’s duality of time and information. Now information is negative entropy, and incremental entropy is proportional to incremental energy, with temperature as the “constant” of proportionality, that is, $dQ = TdS$ where dQ is energy change and dS entropy change. But conventional computation is isothermal, with T held fixed at a few

hundred degrees Kelvin, 300 for slow CMOS, closer to 350 for fast ECL, but in either case not varying significantly in the course of the computation. Somehow physics factors in temperature (as unrelated information competing for limited bandwidth?) in a way our account has not.

This raises the question, is complementarity a feature of quantum mechanics because the universe is basically an information processor, with computation's duality of time and information showing up somehow as the duality of time and energy? We find this highly plausible. Another clue is Birkhoff and von Neumann's quantum logic, which resembles linear logic more closely than say intuitionistic logic in its lack of distributivity of conjunction over disjunction yet satisfying double negation. We are presently looking for more such clues and trying to fit them together into a comprehensive account of the relationship. One goal of this investigation is a less mystical explanation of quantum mechanics than the Copenhagen interpretation of Bohr, which on the one hand has had no really successful challengers in the past sixty years but on the other has left many physicists and philosophers very dissatisfied with the amount of disbelief that must be suspended.

References

- [Bir33] G. Birkhoff. On the combination of subalgebras. *Proc. Cambridge Phil. Soc.*, 29:441–464, 1933.
- [CCMP91] R.T. Casley, R.F. Crew, J. Meseguer, and V.R. Pratt. Temporal structures. *Math. Structures in Comp. Sci.*, 1(2):179–213, July 1991.
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [Joh82] P.T. Johnstone. *Stone Spaces*. Cambridge University Press, 1982.
- [Kle56] S.C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–42. Princeton University Press, Princeton, NJ, 1956.
- [Klo86] Christian Kloesel, editor. *Writings of Charles S. Peirce: A Chronological Edition*, volume 4, 1879-1884. Indiana University Press, Bloomington, IN, 1986.
- [NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures, and domains, part I. *Theoretical Computer Science*, 13, 1981.
- [Pap86] C. Papadimitriou. *The Theory of Database Control*. Computer Science Press, 1986.

- [Pet62] C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress 62*, pages 386–390, Munich, 1962. North-Holland, Amsterdam.
- [Pra90] V.R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI: European Workshop JELIA '90, LNCS 478*, pages 97–120, Amsterdam, NL, September 1990. Springer-Verlag.
- [Pra91a] V.R. Pratt. Event spaces and their linear logic. In *Proc. Second International Conference on Algebraic Methodology and Software Technology, Workshops in Computing*, Iowa City, 1991. Springer-Verlag, to appear.
- [Pra91b] V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.
- [Pra92a] V.R. Pratt. Arithmetic + logic + geometry = concurrency. In *Proc. First Latin American Symposium on Theoretical Informatics, LNCS 583*, pages 430–447, São Paulo, Brazil, April 1992. Springer-Verlag.
- [Pra92b] V.R. Pratt. Origins of the calculus of binary relations. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science*, Santa Cruz, CA, June 1992.
- [Pri70] H.A. Priestley. Representation of distributive lattices. *Bull. London Math. Soc.*, 2:186–190, 1970.
- [Rei85] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.
- [Shi85] M. Shields. Deterministic asynchronous automata. In E.J. Neuhold and G. Chroust, editors, *Formal Models in Programming*. Elsevier Science Publishers, B.V. (North Holland), 1985.
- [Sto37] M. Stone. Topological representations of distributive lattices and brouwerian logics. *Časopis Pěst. Math.*, 67:1–25, 1937.
- [Win86] G. Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, LNCS 255*, Bad-Honnef, September 1986. Springer-Verlag.