A Theory of Conjunction and Concurrency

C.A.R. Hoare

May 1989

Summary

This paper explores some general conditions under which the specification of a concurrent system can be expressed as the conjunction of specifications for its component processes. It proves a lattice-theoretic fixed point theorem about increasing functions, and gives examples of its application in several areas of computing science. Some consequences are drawn for the design of concurrent algorithms, high-level programming languages, and of finegrained concurrent computer architectures

1 Introduction

The natural way to structure a complex specification is as the conjunction of many individual requirements; for example, we would like a widget to be firm and flexible and light and strong and smooth and soft and cuddlesome. How easy widget design would be if we could meet this overall specification by designing seven separate parts, each of which meets only one of the requirements; and then somehow assemble them into a single widget which meets all the requirements simultaneously! And after initial design, how delightful it would be to meet a change in just one of the requirements by changing only the corresponding part of the implementation.

But widget design is not like that, and nor is any other branch of engineering design, not even software engineering. However there are special cases in the design of computer systems, both hardware and software, where a set of requirements can all be met by assembly of a set of components, each of which satisfies only one of them. Furthermore, the method of assembly may permit all the components to operate concurrently with each other. Our goal is to explore some general conditions under which we can take advantage of this remarkable synergy between conjunction of requirements and concurrency of their implementations.

In more mathematical terms, we are interested in a theorem of the form

$$S(f\|g) = (Sf) \text{ and } (Sg)$$

where f and g are designs for components of some product yet to be implemented; $(f\|g)$ denotes the result of assembling them together to operate concurrently; Sf and Sg are descriptions of certain relevant properties which are enjoyed by f and g individually, while $S(f\|g)$ is a description of the properties of their assembly. The equation states that the specification of a concurrent system can be expressed as the conjunction of the specifications for its component processes. Because conjunction is associative, the equation given above generalises to any number of requirements and processes.

2 The mathematical theory

Let us start with a general description of a computational mechanism, equally applicable to hardware or software. Such a mechanism is usually capable of passing rapidly through a series of states. Most of these are of no interest to the user of the mechanism; the only interesting states are those that can be observed, at least in part. Particular interesting states are those in which the mechanism is started, those in which it is waiting for further input, and those in which it stops. Clearly, a description of the intended waiting or stopping states plays an important part in the specification of a mechanism. In this paper we shall concentrate on this aspect, separating it from several other vital concerns.

The behaviour of a simple mechanism can be described by a function d, which maps each possible state of the mechanism onto the next state which that mechanism will assume. When started in state x, the mechanism will go through the sequence of states

$$x, dx, d(dx), \ldots, d^n x, d^{n+1} x, \ldots$$

If at any state further application of d leaves the current state unchanged (i.e., $d^{n+1}x = d^n x$), then all subsequent states will also remain unchanged. This state is therefore called a *stable* state. The set of all possible stable states of the mechanism are the fixed points of the function d, defined:

$$fp(d) \quad \hat{=} \quad \{x \mid dx = x\}.$$

It is also possible that the infinite sequence of iterations of d does not contain one of these fixed points. This may happen in two ways:

(1) oscillation; $d^n x$ equals $d^m x$ only when m differs from n by some multiple of k. In this case, the sequence cycles infinitely often through the same subsequence of k states.

(2) divergence: $d^n x$ equals $d^m x$ only when n equals m. In this case, all members of the sequence occur exactly once.

In both of these cases, the set of fixed points of d fails to overlap with the trajectory of the machine; in the extreme case, the set of fixed points may even be empty. Our theory is valid for this case too, but not very useful. The issues will be explored more fully in the appendix.

Now let us impose a little mathematical structure on the state space of the mechanism. A *partial order* is a relation (written $\leq$) that is reflexive and transitive and antisymmetric, but not necessarily total. We want to use this ordering to compare how close two states are to stability, so that $x \leq y$ means that y is as close or closer to stability than x.

This intended interpretation requires that each application of the transition function d should bring the state closer to stability, or leave it at least as close, i.e.,

$$x \leq dx, \qquad \text{all } x.$$

A function with this property is said to be *increasing*. An immediate consequence is:

Lemma: $dy \leq x \Rightarrow y \leq x$, all x and y.

More significantly, if d and e are increasing functions, so is their functional (sequential) composition $(e; d)$.

Proof:

$$\begin{array}{ll}
 & x \leq d(ex) \\
\Leftarrow \{transitivity\} & x \leq ex \text{ and } ex \leq d(ex) \\
\Leftrightarrow \{d, e\, increasing\} & true
\end{array}$$

□

In order to introduce concurrency we need to make an assumption that every pair of states x and y has a least upper bound $x \vee y$ in the ordering. The least upper bound of an ordering has the property

$$(x \vee y) \leq z \quad \text{iff} \quad x \leq z \text{ and } y \leq z.$$

It is the only state which has this property. From it can be proved that the $\vee$ operator is idempotent, commutative and associative. Furthermore, it is an increasing function in both its operands:

$$x \vee y \geq x \text{ and } x \vee y \geq y.$$

The least upper bound operator can be lifted pointwise to functions over states in the normal way

$$(d \vee e) \;\; \hat{=} \;\; (\lambda x.dx \vee ex).$$

If d and e are increasing functions, so is $(d \vee e)$. Furthermore, it can be computed in parallel by concurrent evaluation of d and e, followed by calculation of the least upper bound. This particular kind of concurrency is extremely effective in cases where calculation of least upper bounds is fast and cheap.

We now prove the fundamental theorem of this paper, that the fixed points of this kind of concurrent composition are exactly the states that are fixed points of both of its components.

Theorem: $fp(d \vee e) = fp(d) \cap fp(e)$.

Proof:

$$\begin{array}{ll}
 & n \in lhs \\
\Leftrightarrow \{def\, fp\} & (d \vee e)n = n \\
\Leftrightarrow \{def \vee\} & dn \vee en = n \\
\Rightarrow \{\vee\, increasing\} & dn \leq n \text{ and } en \leq n \\
\Rightarrow \{d, e\, increasing\} & dn = n \text{ and } en = n, \;\text{ i.e. } n \in rhs. \\
\Rightarrow \{\vee\, idempotent\} & n = (dn \vee en) \\
\Leftrightarrow \{shown\ above\} & n \in lhs.
\end{array}$$

□

But concurrent execution of d and e is not the only way of reaching the required set of fixed points. Sequential composition will do just as well.

Theorem 2: $fp(e;d) = (fp\,e) \cap (fp\,d)$.

Proof:

$$\begin{array}{lll}
& \{assumption\} & n \in rhs \\
\Leftrightarrow & \{def\ fp\} & dn = n \text{ and } en = n \\
\Rightarrow & \{Leibnitz\} & dn = n \text{ and } d(en) = dn \\
& \{transitivity\} & d(en) = n \qquad\qquad \text{i.e. } n \in lhs. \\
\Rightarrow & \{d\ increasing\} & en \leq n \\
\Rightarrow & \{e\ increasing\} & en = n \qquad\qquad (1) \\
\Rightarrow & \{Leibnitz\} & d(en) = dn \\
& \{n \in lhs\} & d(en) = n \\
\Rightarrow & \{trans\ =\} & dn = n \qquad\qquad (2) \\
\Rightarrow & \{(1),(2)\} & n \in rhs.
\end{array}$$

□

A consequence of this is that the intersection of the fixed points of d and e can be implemented by any combination of sequential or parallel composition, e.g.,

$$((d;e^2) \vee (e;d));d^4$$

The choice between these combinations can be made, perhaps even while the mechanism is running, to increase the likelihood of rapid convergence.

It also has important consequences for the design of algorithms to exploit concurrency, of programming languages to express them, and of machine architectures to implement them. Suppose a program is specified by describing a set S of desired stable states. Suppose the program is expressed in a language like UNITY [] as a set $\{d, e, \ldots\}$ of functions. It can be proved correct simply by showing that each function is increasing in some appropriate order, and that

$$S = fp(d) \cap fp(e) \cap \ldots$$

Now a sequential machine architecture can implement the program by iterating the functions $d, e, \ldots$, interleaving them in any way which avoids total neglect of any one of them.

However, a parallel machine architecture can be much more optimistic; it can embark on simultaneous evaluation of *all* the functions. As soon as *any* of them is ready to deliver a result, the least upper bound of this result and the current state is computed, and the same function is rescheduled for later calculation. If however (say as a result of collision or resource limitations or page breaks or hardware failure) it is impossible on this iteration to take advantage of a particular calculation, *it is just discarded.* The function will anyway be recomputed at some later time. There is no need to resort to expensive techniques like backtracking and sequential execution, which seriously complicate and slow down the normal "optimistic" scheduling of fine-grained concurrency []. In spite of the combinatorial explosion of non-determinacy in the approach to the fixed point, the determinacy and (what is more important) the correctness of the final result is guaranteed. That is why this kind of non-determinacy is said to be *confluent.*

2.1 Examples

The first example is due to Chandy and Misra, who used it as a vivid introduction to the concepts of UNITY [].

Let F be the set of currently free hours marked in the diary of a busy professor. F will be represented as a set of natural numbers. F serves as the specification of a function which maps an arbitrary hour h onto the next subsequent hour at which the professor is free, or to h itself if this is free. The function is one that is often computed in response to a telephoned enquiry.

One implementation of it is as follows. Initially, point your finger to hour h in the diary. If this hour is free, then give h itself as the answer. Otherwise move the finger on to the next hour and repeat. The transition function of this simple mechanism is the function

$$d = \lambda h.\ \textbf{if}\ h \in F\ \textbf{then}\ h\ \textbf{else}\ h+1$$

The proof that it meets the specification F is simply that (obviously) $fp\,d = F$; the proof that it is increasing is equally simple. Of course, if F is finite then iteration of d may fail to terminate; but that is of no concern to our theory, which has chosen to ignore the problem of convergence to fixed point.

Now let G specify the set of free hours of a second professor, and let e be any implementation of it. Suppose we wish to arrange a meeting between these two professors. Clearly, such a meeting can take place only at an hour at which both professors are currently free, i.e., in the intersection $F \cap G$. This therefore serves as the specification of a function, to be implemented with the aid of the functions d and e, which implement F and G respectively.

Our fundamental theorem states that one possible implementation is the function $(d \vee e)$. This function is in practice often computed over the telephone. Each professor computes the hour at which they are next free, and communicates it to the other. If they have given the same hour, this is the right answer. Otherwise, the later of the two times is selected by both of them, and the process is repeated. Eventually, the process gives the earliest hour at which the two professors can meet. Meetings between three or more professors can be arranged with an even higher degree of concurrency and intermediate non-determinacy.

A Unity program which expresses the algorithm is

```
Program
assign    x := dx  []  y := ey  []  x := y if y ≥ x  []  y := x if x ≥ y
end
```

2.2 Deterministic data flow networks [].

A process network consists of a number of concurrently active nodes, connected by named single-directional channels. The state of each channel is represented by the sequence of messages that have been output to it since the network first started. The state of the whole network is represented by a function n which maps each channel name x to its current state nx. In the initial state of the network all channels are empty:

$$nx = \langle\ \rangle, \qquad \text{for all channels } x.$$

A node in the network is specified by the set of its stable states. Each of these is a function from the names of the channels incident upon the node to the sequence of messages that have passed along them. A state is stable for a node if the node is waiting for input, and can make no further progress until it arrives. A transition function for a node is one that maps each stable state to itself; but each non-stable state is extended by outputting a message on an output channel of the node. By definition of stability, this is always possible. Of course, such a minimal transition can be iterated up to (but not beyond) the next higher stable state for that node. The fact that the transition function for each node is a function ensures that the whole network will be deterministic.

Note that a transition function can only extend one or more of the values of each channel name, so that the value of each channel before the transition is always a prefix (initial subsequence) of the value after. This is true of each channel individually; it is therefore true of the overall state of all channels together. We therefore define an ordering over states as the pointwise extension of the prefix ordering over sequences:

$$n \leq m \quad \text{iff} \quad (nx \leq mx \text{ for all channels } x.)$$

In this ordering, each transition function is an increasing function, as required by our theory.

When two nodes (or even subnetworks) are assembled together, the channels with the same name within each subnetwork become connected; and so in the combined state the channel with that name must take a single value, which is a function of the possibly different values ascribed by the two components. In order to apply our theory, this function should be the least upper bound in some ordering which makes the individual transition functions increasing, i.e., the prefix ordering as defined above on network states.

Unfortunately, in the prefix ordering the least upper bound of two sequences does not exist unless one of the two is a prefix of the other. In order to ensure the existence of least upper bounds, we have to rely on the following two properties of data flow networks:

(1) Each channel is single-directional and has only one node which is permitted to extend it by outputting a message. Thus the current value of each channel is determined solely by the state of the subnetwork which contains the outputting node.

(2) Each node is always prepared to accept any message on any of its input channels. Thus each node and subnetwork is always prepared to accept any lengthening of the current values on its input channels, without changing its output behaviour. This is usually implemented by putting unbounded buffers on each input channel.

Thus the defining characteristics of a deterministic data flow network are just those which ensure the existence of the least upper bounds which make our theory applicable.

2.3 C-mos circuit design.

A C-mos circuit consists of a number of transistors connected by named wires, which are inherently bidirectional. In our simplified model, the state of each wire is represented by a pair of measurements (h, l), which are scaled so as to range between zero and one. The value of h is a measure of the strength of the connection of the wire to a source of high voltage (power), and l is a similar measure of connection to a source of low voltage (ground). Both measures are taken along the path of least resistance.

Four of these states are of particular interest. $(1, 0)$ and $(0, 1)$ represent the logic levels "true" and "false" respectively. These values are said to be *driven*, and they are the most useful states for computational purposes. The state $(0, 0)$ is called floating; it is totally disconnected from both power and ground, and has a correspondingly neutral value. The worst state is $(1, 1)$, because it indicates a short circuit, i.e., a perfect connection between power and ground along which current can flow unimpeded. If this state is stable, the whole circuit will fail (due to heating, metal migration, etc). In fact any state (h, l), with both l and h greater than some positive threshold, is undesirable in C-mos design.

Now suppose a wire in state (h, l) becomes perfectly connected to a wire in state (h', l'). If l is less than l' then the path of least resistance to ground passes through l'; so this is the new measure of the strength of connection to ground of the resulting single wire. A similar argument applies for connection to power. So the whole state of the connecting wire and both its ends is $(h \vee h', l \vee l')$, where $\vee$ gives just the greater of its two operands. This is nothing but the least upper bound in the standard product ordering over the interval zero to one:

$$(l, h) \leq (l', h') \quad \text{iff} \quad (l \leq l' \text{ and } h \leq h').$$

A single wire must be the cheapest and fastest possible way of computing a least upper bound; and this explains the fundamental role of wires in the implementation of computers by hardware.

As in the case of data flow networks, the state of a whole circuit is given by a function from wire names to the state of each wire. When two circuits are connected, wires with the same name become identified, and assume a value equal to the least upper bound of that ascribed by the separate subcircuits. We can therefore apply our fundamental theorem that the stable states of the whole circuit are just the intersection of the stable states of its components, and ultimately those of its individual transistors.

A transistor is connected into a network by three wires, known as its gate (g), its source (s) and its drain (d). It acts as a variable resistance between its source and its drain; where the variation is controlled by the value at the gate. The stable states of the transistor may be described in terms of certain thresholds. For example, if the gate of an N-transistor is in state $(1,0)$ and the source is in state $(0,1)$, then the transistor acts pretty well like a perfect connection. As a result, when stability is reached the drain will also be in state $(0,1)$. The argument is symmetric between the source and drain wires. Thus any stable state of the N-transistor will satisfy the predicate

$$\text{if } g = (1,0) \text{ then } (s = (0,1) \quad \text{iff} \quad d = (0,1))$$

If d is connected to the ground (ie., a wire which always has value $(0,1)$), this simplifies to

$$\text{if } g = (1,0) \text{ then } s = (0,1).$$

Similarly, consider the stable states of a P-transistor with gate g, source s and drain permanently connected to the power wire, which has value $(1,0)$. These satisfy the property

$$\text{if } g = (0,1) \text{ then } s = (1,0).$$

If this P-transistor is connected to the N-transistor described above, we get a conventional negation circuit with input g and output s. Our fundamental theorem establishes that its stable states satisfy the specification of a negation circuit, namely:

$$\text{if } g \text{ is driven then } (s = \neg g \text{ and } s \text{ is driven })$$

where "g is driven" means $g \in \{(0,1),(1,0)\}$
and $\neg(x,y)$ means (y,x).

Unfortunately, the transition function of a transistor is not an increasing function in our chosen ordering of wire values. However, in the region of $(0, 1)$ and $(1, 0)$ it behaves sufficiently like an increasing function; and in other regions we can describe its behaviour by a theory that makes it seem *worse* than it actually is. Any designs based on the theory can then be validly implemented by hardware; although the implementation can behave differently from what is predicted by theory, this happens only when its actual behaviour is better. No greater accuracy is required of a theory intended for use in engineering design.

2.4 Logical inference.

A rule of logical inference may be regarded as a function d from a set of sentences (its hypotheses) to another set of sentences (its conclusions). The result includes all sentences which can be deduced by a *single* application of the rule of inference to *any* subset of the hypotheses which match the antecedent of the rule. Without loss of generality, we may repeat all the hypotheses among the set of conclusions. As a consequence

$$S \subseteq dS$$

and d is an increasing function in the inclusion ordering between sets. The fixed points of d are just the sets of sentences (theories) which are deductively closed under the inference rule.

Now if e is another inference rule, the deductive closure of the combination of the rules d and e can be computed by two concurrent mechanisms evaluating d and e separately, provided that each of them occasionally updates its current state by including the set of sentences currently in the state of the other, i.e., the least upper bound in the inclusion ordering.

Of course, most deductive closures have an infinite set of sentences; and no computation can ever produce an infinite set as a result. So iteration of the functions d and e described above never reach their fixed points. However, they approach arbitrarily close, in the sense that every sentence of the closure is eventually discovered, and included into the set. However even these brief remarks violate our initial resolve not to mention the question of convergence. The point will be treated properly in the appendix.

An interesting example of deductive closure is given by the computation of fractal diagrams. Figure 1 shows such a diagram, a Sierpinski triangle. It is an equilateral triangle with removal of the triangle formed by connecting the midpoints of the three sides, and with the same excision made recursively to each of the remaining three sub-triangles. The set of points remaining after all these excisions form the Sierpinski triangle. As a result of the recursion, the whole triangle is isomorphic to each of the three subtriangles in its corners. The three isomorphisms (in the co-ordinate system shown in Figure 1) are f, g, h, defined thus:

$$\begin{aligned} f(x, y) &= (x/2, y/2) \\ g(x, y) &= ((x+1)/2, y/2+1) \\ h(x, y) &= ((x-1)/2, y/2+1) \end{aligned}$$

Each of these functions can serve as an inference rule, giving rise to a transition function

$$\lambda S.(S \cup fS)$$

and similarly for g and h. The points of the Sierpinski triangle can be enumerated by concurrent computation of these three transition functions, starting with the set containing only the origin (0,0).

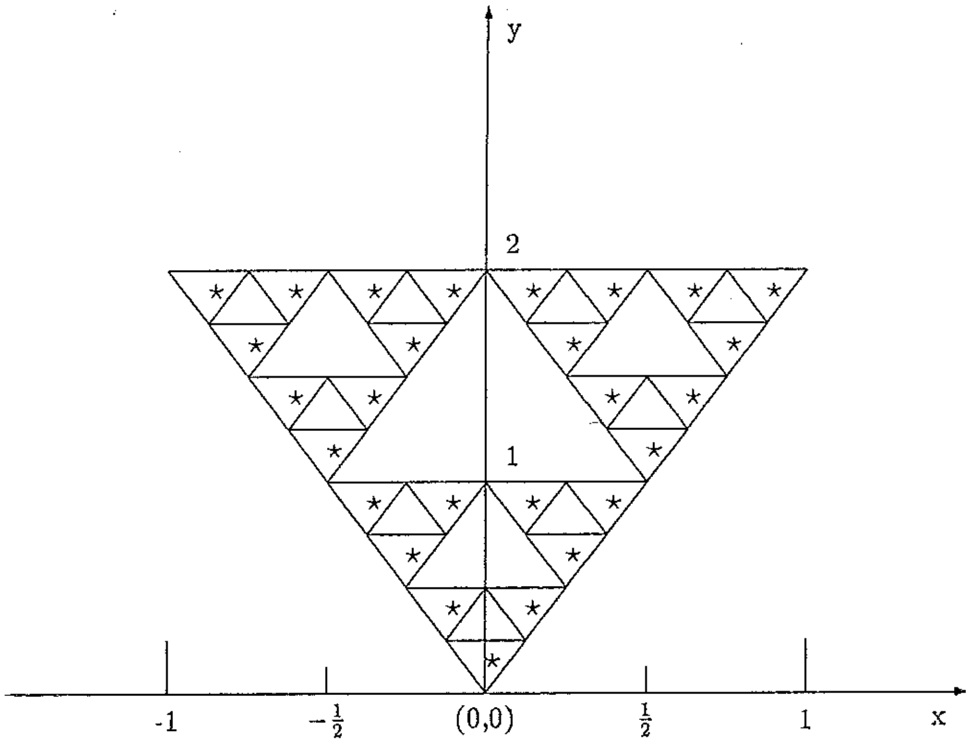

Figure 1: Sierpinski triangle

A Appendix

This appendix discusses some extensions and variations of our theory, which may be useful in many applications.

A.1 Monotonicity of transition functions

A function d is said to be *monotonic* in a given ordering $\leq$ if

$$x \leq y \Rightarrow dx \leq dy \quad \text{for all } x \text{ and } y.$$

This is a reasonable condition to impose upon a transition function if we wish to use the ordering to compare two states for closeness to stability: it ensures that when one mechanism starts closer to stability than another, it will remain so after each mechanism has progressed the same number of steps. The further stipulation that all transition functions should be monotonic as well as increasing does not invalidate our theory as expanded so far. This is assured by the theorem.

If d and e are monotonic, so are $(d; e)$ and $(d \vee e)$.
Nor does it invalidate any of our examples, all of which are monotonic.

Monotonicity has the further surprising benefit that it determines precisely which of the fixed points of d (if any) will be reached by a mechanism started in state x. The terminal state is

$$\begin{array}{ll} & min((fp\, d) \cap (x\uparrow)) \\ \text{where} & x\uparrow = \{y \mid x \leq y\} \end{array}$$

and $min\, X$ is the least member of X, a property uniquely defined by the law

$$(min\, X) \in X \text{ and } (min\, X) \leq y, \text{ all } y \text{ in } X.$$

This is assured by the theorem

$$\text{If } d^n x = d^{n+1} x \text{ then } d^n X = min((fp\, d) \cap x\uparrow).$$

Proof:

$\{antecedent\}$	$d^n x = d^{n+1} x$	
$\{def\ fp\}$	$d^n x \in fp\, d$	
$\{d\ increasing\}$	$x \leq d^n x$	
$\{set\ theory\}$	$d^n x \in (fp\, d) \cap (x\uparrow)$	(1)
$\{assumption\}$	$y \in (fp\, d) \cap x\uparrow$	
$\{def\ x\uparrow, fp\}$	$x \leq y = dy$	
$\{d\ monotonic\}$	$dx \leq dy = y$	
$\{induction\ on\ n\}$	$d^n x \leq y$	
$\{discharge\ assumption\}$	$d^n x \leq y$ all y in $fp\, d \cap x\uparrow$	
$\{(1), uniqueness\ of\ min\}$	$d^n x = min((fp\, d) \cap (x\uparrow))$.	$\square$

A requirement of monotonicity of the transition function greatly strengthens our claim that a mechanism is adequately specified by its desired set of fixed points.

A.2 Preorder instead of partial order

A preorder $\leq$ is defined as a relation that is reflexive and transitive, but not necessarily antisymmetric. Two states are said to be *equivalent* if each is related by the preoreder to the other

$$x \equiv y \text{ means } (x \leq y \text{ and } y \leq x).$$

This is an equivalence relation, expressing the fact that two possibly differing states are equally close to stability. If d is an increasing function, then all the states in an oscillating cycle are equivalent to each other. They are all of them fixed points, according to the appropriate weaker definition which uses equivalence in place of equality

$$fp\, d = \{x \mid dx \equiv x\}.$$

The replacement of a partial order by a preorder requires other similar replacements of equality by equivalence in definitions theorems and proofs. Apart from this, the whole theory remains valid, provided that transition functions are required to be monotonic. This is because monotonic functions respect equivalence

$$\text{if } x \equiv y \text{ then } dx \equiv dy.$$

This is another good reason for insisting on monotonicity.

The advantage of weakening the partial order to a preorder is that it widens the range of applicability of the theory, for example to include oscillating systems.

A.3 ω-completeness and approximation.

The least upper bound $\bigvee S$ of a subset S is a preorder is the defined up to equivalence by the property

$$\bigvee S \leq y \text{ iff } (x \leq y, \text{ for all } x \text{ in } S).$$

The existence of a least upper bound may depend on some property of the set S, for example that it is a countable increasing chain $\{x_o, x_1, \ldots\}$, where $x_i \leq x_{i+1}$ for all i. If all such increasing chains have a least upper bound, the preorder is called ω-cocomplete (or more usually ω-complete, for short). From now on, we assume our preorder has this property.

Now if d is increasing, the iteration sequence $\{x, dx, \ldots, d^n x, \ldots\}$ is an increasing chain, and therefore has a limit denoted $\bigvee_n d^n x$. If in fact iteration of d reaches a fixed point, then this is exactly the least upper bound. But in the other case (divergence), the least upper bound is a sort of infinite element, to which the iteration approaches arbitrarily close but can never reach. An infinite element is a mathematical figment, since all our actual mechanisms are inescapably finite. For this reason, we are free to define the result of applying a transition function d to an infinite element in any way we like; and obviously what we like most is to make it a fixed point of d, ie.,

$$d(\bigvee^n x) \equiv \bigvee(d^n x)$$

With this convention, our entire theory applies equally well to infinite as to finite fixed points; and so covers such cases as logical theories and fractal images. But in applications where infinite states are not desired and not needed, the theory works equally well without them.

$[\![\exists X . A]\!] = \lambda c_0 .$

$\exists X A \;\; \exists X A$

$(X=1) \;\; (\exists X . (X=2)^* \to nil) \;\; x=2 \;\; true$

$f_A(c_0) = c_1$

$\overset{d}{=} \;\; \langle A, c_0 \rangle \longrightarrow^* \langle A', c_1 \rangle = P$

P is terminal

$$\left\{ \begin{array}{ll} ① & f(c) \Rightarrow c \\ ② & f^2(c) = f(c) \\ ③ & c_0 \Rightarrow c_1 \quad \text{imp} \quad f(c_0) \Rightarrow f(c_1) \end{array} \right.$$

$[\![NIL]\!] = I$

$[\![c* \to A]\!] = \lambda c_0 . \; [\![A]\!](c \wedge c_0)$

$[\![c\downarrow \to A]\!] = \lambda c_0 . \; \underline{\text{if}} \; c_0 \Rightarrow c \; \underline{\text{then}} \; [\![A]\!](c_0) \; \underline{\text{else}} \; c_0$

$\exists$

$[\![A_1 \| A_2]\!] = \lambda c_0 . \underline{\min} \; [\![A_1]\!]^* \cap [\![A_2]\!]^* \cap \{ c \mid c \Rightarrow c_0 \}$

$[\![\exists X . A]\!] = \lambda c_0 . (c_0 \wedge \exists X . ([\![A']\!](\exists X . c_0)))$

$[NIL] = true$

$c* \to A = c \wedge A$

$c\downarrow \to A \triangleq c \Rightarrow A$

$A_1 \| A_2 \quad A_1 \wedge A_2$

$\exists X . A = \exists X . A \Rightarrow A(X / \min_x . A)$