Tony,

A lovely introduction: presumably something very like this will appear in the 1st year logic course next year.

One comment: I think that throughout you ought to say a chain <u>which is bounded above</u> (ie, has an upper bound) has a least upper bound. The underlined phrase seems to be missing entirely from both the formal (ie appendix) and informal treatments.

I've made minor comments on a photocopy.

Jeff.

written on sabbatical in Austin Texas

Introduction to Domain Theory

never submitted

C.A.R. Hoare

March 1987

Summary. A recursive program unit x is one which is defined to be equal to f(x), where f(x) is expressed in the notations of some programming language, and contains (recursive) occurrences of the name x. Domain Theory is the branch of mathematics which studies general conditions under which equations of the form x = f(x) have a computable solution. Since recursion is the most general method of writing short programs to evoke long calculations, Domain Theory is fundamental to a study of Computing Science.

Recursion in Programming

The purpose of a computer program is to control the calculations of a general-purpose high-speed computer. Usually we want the program to be very much shorter than the calculations which it evokes. For this reason, we use a programming language which permits the definition of component units of a program by some kind of recursion. A recursive program unit X is defined by a program text which actually contains its own name X. In general, mathematics forbids such circular definitions; but if the formula is expressed wholly in the notations of a programming language, occurrences of the defined name on the right hand side of the definition are executed as recursive calls on the whole program unit, and this leads to calculations of arbitrary length.

any desired

The simplest example of recursion is the program loop. Let b be the condition which controls loop termination, and let P be the program text to be executed on each repetition. The whole loop X can be defined by the equation

X = **if** b **then begin** P ; X **end**

If b is false, X terminates immediately. If b is true, execution of P is

followed by execution of the whole of X again. In a high level programming language, this special case of recursion is usually hidden by a specialised notation such as

while b **do** P

and it is implemented very efficiently by a conditional and an unconditional jump:

```
startloop : if not b then goto endloop;
            ...P...;
            goto startloop;
endloop   : . . . .
```

The efficiency of this implementation depends on the fact that the recursive call on X occurs only at the end of the recursively defined program unit. This is a special case of recursion known as tail recursion. In general, the recursive call may occur in the middle of the program unit, perhaps several times, for example

X = **if** b **then begin** X; P; X; Q **end**

The efficient implementation of this more general form of recursion on a computer requires an ingenious combination of jumps, links, stacks and pointers. Disregarding efficiency, it is easier to explain how recursive programs are executed by the following crude but simple technique, which was used to explain the meaning of recursion in ALGOL 60 and has occasionally been used to implement it in FORTRAN. In the program fragment F(X), which defines the behaviour of the recursive unit X, replace every recursive call on X by a command to print an apology and then stop

print "sorry, it didn't work"; stop;

The result of the substitution has the form

F(**print** apology; stop)

Submit the modified program together with its input data for execution on your computer. With a great deal of luck it will actually work, because on this occasion and for this particular input data there was no need for the computer to make any recursive call on X, so it also bypasses the print

all occurrences of

statement which has replaced X. But if an apology message appears, take a copy of the whole program that has just failed, substitute it for X in a new copy of F(X), and then submit this much longer program for execution. Repeat this process of substitution and resubmission as often as necessary, until the the program runs right through without giving its message of apology. If this happens after say 37 steps, the program which was actually executed on this occasion was the result of 37 substitutions

F(F(F(. . . . . (F(**print** apology ; stop)) . . . . .)))

37 times

In the case of unsuitable input data, or (more likely) a programming error, the process of resubmission will never reach a successful conclusion: one of the reasons for studying the mathematical properties of recursion is to learn how to avoid this mistake.

Recursion is useful not only in the specification of lengthy calculations. Programming languages themselves are also large and complex, and and recursion is needed to describe ~~not only~~ their notations (or syntax), but also their meaning (or semantics). The data structures stored in computers and manipulated by programs are also very large and complex, and they too require recursion for their definition, description and control. The value of recursion as a means of understanding and controlling complexity lies in its great simplicity and generality. It has a clear mathematical meaning and simple mathematical properties, which are explored in a branch of mathematics known as Domain Theory. The value of Domain Theory for Computing Scientists is that it strengthens our understanding of recursion and our ability to use it effectively.

Recursion in Mathematics

Now let us see how the idea of rcursion appears in more traditional branches of mathematics. ~~Suppose there is given some~~ Consider a mathematical function g, for example cosine or square root The equation

$$x = g(x)$$

may be regarded as an equation in one unknown x, whose solution can be sought by ~~some mathematical technique designed for this purpose require of~~ one of the many techniques provided by mathematics for the solution of equations.

Any solution to this particular equation is called a fixed point of the function g because it remains unchanged when you apply g to it. As an obvious consequence, it remains unchanged when you apply g to it twice or any number of times, including no times at all.

Fixed points are very familiar in mathematics. For example, zero is the only number which remains the same when you negate it. So zero is the fixed point of the function which multiplies its argument by minus one. It is also a fixed point of halving, doubling, squaring, and taking the square root. Here is a less trivial example. Consider the function g of real numbers which adds a half of its argument to the reciprocal of its argument. It is defined formally by the simple non-recursive equation

$$g(x) = x / 2 + 1 / x$$

Let us apply this function to the square root of two

$$\begin{aligned} g(\sqrt{2}) &= \sqrt{2} / 2 + 1 / \sqrt{2} \\ &= (\sqrt{2} + \sqrt{2}) / 2 \\ &= \sqrt{2} \end{aligned}$$

So the square root of two is a fixed point of this function g.

Now suppose we wish to calculate the numeric value of the square root of two. The Newton/Raphson method just applies the function g repeatedly to some arbitrary starting value, say the number one. Here are the first few steps

$$\begin{aligned} g(1) &= 1/2 + 1/1 &&= 3/2 \\ g(g(1)) &= (3/2)/2 + 1/(3/2) = 3/4 + 2/3 &&= 17/12 \\ g(g(g(1))) &= 17/24 + 12/17 &&= 577/408 \end{aligned}$$

Let us stop here and square each of these numbers:

$$\begin{aligned} g(1)^2 &= 2 + 1/4 \\ g(g(1))^2 &= 2 + 1/144 \\ g(g(g(1)))^2 &= 2 + 1/332928 \end{aligned}$$

If we continue applying g again and again, we get a series of fractions which get nearer and nearer to the square root of two, but can never reach it,

because the square root of two is not a fraction. However, we can get as close as we like by applying g any number of times; and if we find we haven't got close enough, we can apply it yet again. So by applying the function again and again, we have obtained an infinite sequence of numbers, each of which is better than its predecessor, and which can approach indefinitely close to the true fixed point, i.e. the square root of two, which satisfies the equation

series

$$x = g(x)$$

This mathematical method of computing a fixed point by repeated application of a function is very similar to the crude technique of substitution used in my previous explanation of recursion in computer programs, ~~where~~ each substitution is at least as likely to succeed as its predecessor, and in some circumstances perhaps will succeed even when its predecessor has proved inadequate. Domain Theory is the branch of mathematics which studies general conditions under which the fixed point of a function can be computed or approximated in this way by repeated application of that function to some suitable starting value. That is why Domain Theory is relevant to a proper understanding of recursion in Computer Science.

successive

Domain Theory

But life is not always as easy as it was for the simple examples I have chosen so far.

An attempt to calculate the fixed point of a function by applying it again and again may go wrong in three different ways:

1. There may actually be <u>no</u> fixed point for the function. In a conventional number system, there is no number which remains the same when you add one to it. So the successor function, which adds one to its argument, has no fixed point. Domain Theory ~~tells us~~ how to design programming languages so that all recursive definitions have a fixed point.

solves this problem by showing

2. There are functions which have fixed points, but these cannot be approached by iteration. For example zero is the fixed point of the negation function, which multiplies its argument by minus one. But if we try to compute this fixed point by iteration starting with the number one, we obtain the series

1, -1, 1, -1, 1, ...

and we never get anywhere nearer the true fixed point we want, namely zero. The same thing happens if we start with any number other than the fixed point zero itself. Domain Theory provides a single universally suitable starting point which ensures that this kind of non-convergence will not occur.

3. Finally, there are functions which have more than one fixed point. For example the square of zero is zero and the square of one is one. So both zero and one are fixed points of the squaring function. For such cases Domain Theory allows us to specify in advance which of the alternatives we want. It is this third problem that we will solve first. Let us allow the selection between alternative solutions to be made in accordance with some fixed scale of comparison always giving the fixed point which is the lowest (or least) in this scale. But the scale must be consistent. We cannot allow two different points each to be strictly lower than the other. A consistent scale of comparison is known as a partial order, in that it enjoys the following three properties

(1) every point is as low as itself (the reflexive property).

(2) if some point is as low as another, and this other point is as low as some third point, then the first point is as low as the third (the transitive property).

(3) there is no distinction between points that are as low as each other (the antisymmetric property).

Domain Theory does not require us to specify the order to be a <u>total</u> ordering. There may be two points of which we do not want, or are not able, to say whether or not one is as low as the other. Such points are said to be <u>incomparable</u>. For example, a racing skier may regard one point on a mountain as lower than another if he can reach the lower point from the higher point by a continuous ski run steeper than thirty degrees. Two different points at roughly the same height on the same mountain are incomparable, because there is not enough slope to enable a racing skier to reach one from the other. And two points on different mountains separated by a valley are always incomparable because one cannot ski across the valley. In spite of the fact that points may be incomparable, Domain Theory will guarantee that among the fixed points of a function there will always be a lowest point, which will be selected in accordance with our chosen scale of comparison.

Consider now the identity function, which maps everything onto itself. Its fixed points are solutions of the equation $x = x$. Since absolutely everything satisfies this equation, the value selected as the solution of this vacuous equation must be lower in our scale of comparison than everything else whatsoever. Such a solution is called the bottom of the partial order. It is easy to prove that there is only one point with the property of being as low as or lower than every other point. For if there were two such points, each of them would be at least as low as the other, and by the antisymmetric property of our partial order, there is no distinction between points which are as low as each other.

Both in mathematics and in computing we are interested in definitions and formulae which generate an unbounded series of results. If every member of a series is as high as or higher than its predecessor, we will call the series a chain. Of any two members of a chain, one is higher than the other, so a chain contains a subset of the partial order which looks like a total order. According to our definition, any series with only one member is a chain, and if one point is lower than a second, the series containing just these two points is a chain. Finally, a chain remains a chain on removal of any number of its elements, but not all of them.

Any finite chain has a maximum element, namely the last element of the series. But an infinite chain does not have a last element, so it does not have a maximum, except in the case that it ends with an infinite series of copies of the same value, for example

1, 3, 5, 7, 7, 7, . . .

A simple example of a numeric chain with no maximum is the series of fractions

1, 1 1/2, 1 3/4, . . ., $2 - 1/2^n$, . . .

If you choose any member of this chain, there is always a higher one, and then a higher one still. Each successive number is closer to the number two, and you can get as close as you like. But the number two does not appear in the chain, and you can never reach it.

However, the number two can be reached by the following more subtle strategy. Consider any real number greater or equal to all members of this chain of fractions, for example 3 or pi or 12.7. Such a number is called an upper bound of the whole chain. Take the set of all such upper bounds, and ask the question whether this set has a minimum member. In fact it does - namely, the exact number two. This is the least of all the upper bounds of the chain we started with, so it is called the least upper bound or lub of the chain. The general strategy, if a chain has no maximum member, is to look instead for a minimum point among all its upper bounds . In the case of real numbers, this strategy is certain to succeed, provided that the set of upper bounds is non-empty. The least upper bound is a good generalisation of the concept of a maximum, since if a set contains a maximum element, this is also its least upper bound; and the least upper bound has most of the mathematical properties of the maximum.

All the definitions given above are summarised in a single definition, that of a domain. A domain is defined as a set with the following three properties:

(1) Its members are ordered by a partial ordering

(2) It contains the bottom point in that ordering

(3) Every chain with elements drawn from the domain has a least upper bound in the domain (but not necessarily in the chain).

The purpose of the partial order is to define a scale of comparison for selection between solutions of an equation when more than one such solution exists. But in fact the same ordering also gives a method of solving the other two problems mentioned earlier. It gives an easy criterion for deciding whether a function has a computable fixed point, and for defining new functions with the same property. It also guarantees that the desired fixed point can be calculated by the method of iteration. These two topics will be treated in the remaining sections of the paper.

Functions with fixed points

Our first task is to find a sufficiently large and useful class of functions which are certain to have a fixed point. A promising class of candidates are those functions which tend to give larger results when they are applied to

larger arguments. More accurately, a function is defined to be monotonic if it commutes with the operator which takes the maximum of a finite chain. In other words, if you apply the function to the maximum of any finite chain you get the same answer as if you apply it to each member of the chain, and then take the maximum of the results. For example, on non-negative real numbers the operation of taking the square is monotonic: if you want the largest of the squares of numbers in a set, you only need to choose the largest member of the set and square it. Other monotonic functions on non-negative numbers are square root, cube root, and truncation, which gives the greatest whole number equal or less than its real argument. For example the truncation of the square root of two is one, and so is the truncation of one itself. But functions like sine and cosine are not monotonic, because they go down as well as up when their argument increases.

A monotonic function is one that preserves the ordering of its operands: if you heighten the operand you heighten the result - or at least you leave it unchanged. Here is a proof. Consider any chain with just two elements. Call the lower of them x and the higher y, so that y is the maximum of the chain. Apply f to both x and y. Because f is monotonic, f(y) is the maximum of f(x) and f(y). So f(x) and f(y) are ordered in the same way as x and y. ~~An important consequence is~~ that when a monotonic function is applied to every member of a chain, the result is still a chain.

Any new functions defined in terms of other monotonic functions will also be monotonic. This fact is important because it means that programming language designers have to make sure only that the built-in facilities of the language are monotonic in the domain ordering. Then any program written in the language will also be monotonic in the sense required to allow recursion to work successfully.

The definition of a monotonic function needs to be adapted to deal with infinite chains which may have a least upper bound but no maximum. A function is defined to be continuous if it ~~distributes over~~ the operation of taking the least upper bound of any chain. Since all maxima are least upper bounds, all continuous functions are also monotonic, and so preserve the ordering of their operands. For example, on non-negative real numbers squaring, square rooting, and taking the cube root all satisfy this definition of continuity. However, the truncating function, which takes each integer to itself and each non-integer to the next lower integer, is not continuous, even though it is monotonic. This is simply shown: consider the same chain as before

C = 1, 1 1/2, 1 3/4, . . .

If you truncate each member of this chain, you get a chain containing only the number one:

trunc(C) = 1, 1, 1, . . .

and the least upper bound of this is also one

lub(trunc(C)) = 1 .

But if we first take the least upper bound of the original chain we get two; and when you truncate two you get two. This is not the same as when we performed the two operations in the reverse order

lub(trunc(C)) = 1

trunc(lub(C)) = 2

This example shows the importance of the concept of continuity

Scientists and engineers prefer to avoid such discontinuous functions, in the hope that they do not occur in nature. Designers of Programming Languages should also avoid such functions if they wish to allow free use of recursion.

The proof that every continuous function f has a fixed point depends on a construction which actually defines the required fixed point as the least upper bound of a chain of approximations, ~~which~~ in general are not themselves fixed points, but provide a path leading up to the fixed point, in the same way as the square root of two is approached by a series of approximations in the Newton/Raphson method. The path starts at the very bottom point in the domain. The next point up the path is obtained by applying the function f to the previous point. After n steps, the function f has been applied n times to the starting point at the bottom. We shall prove shortly that this path is a chain, in that no step can lead downwards. So there are only two possibilities. It may be that after a certain number of steps the next application of f leaves you in the same place as before. All further steps will lead you back to this same point. So this point is already a fixed point of the function f . ~~That is the bad news, because it corresponds to a recursive program that will never terminate. The good news is that~~ the path may go on winding up forever, without ever reaching a fixed point. But now we get the reward for all the hard mathematics we have

they

These approximations

In domain theory

or the results of a recursive program unit are computed by the method of successive substitution

Alternatively

done. Coiling ourselves for one great leap we can jump right over all the infinite number of points of the path, straight to its least upper bound. Provided that the path is a chain, our basic definition of a domain guarantees the existence of this lub; and it turns out to be not only a fixed point of the function, but among all the fixed points it is the one we want, namely the least one.

Proof of the correctness of the construction falls into three parts:

1. we must show that each point on the path is at least as high as the previous one. This ensures that the path is a chain, so that the least upper bound exists.

2. we must show that application of f to the least upper bound of the path gives back the same result.

3. we must show that every other solution of the equation x=f(x) is higher than the least upper bound of the set we have constructed.

1. To prove that all steps on the path lead upwards, we first show that the first step leads up; and then for any step we prove that the next step leads up, provided that the previous step did. By mathematical induction it will follow that all steps lead upwards.

1.1. The first step starts at the very bottom of the whole domain. No step can lead down from there. So the first step leads up. That was easy.

1.2. Suppose the previous step took us up from point x to point fx. Then the next step takes us from fx to f(fx). The start point of the next step has been obtained by applying f to the start point of the previous step, and the end point of the next step has been obtained by applying f to the end point of the previous step. Since f is monotonic it preserves the ordering of its arguments. Since we can assume that the previous step was upward, we know the next step is upward too. By induction, all steps are upward, and the path is a chain.

2. The second of our three tasks is to show that the least upper bound of the path is a fixed point of f. So lets see what happens when you apply f to it.

Since f is continuous, when you apply f to the least upper bound, you will get the least upper bound of the result of applying f to the individual points in the path. But each time we apply f to a point on the path we just get the very next point on the very same path. As a result, we get back every point on the path, except possibly the very first point, which is the bottom of the whole domain. So the resulting path is also a chain. To add or remove the bottom point does not change the least upper bound of a chain. So when we apply f to the whole path, we remove its bottom perhaps, but we do not change its least upper bound. That proves the most important theorem, that all continuous functions on a domain have a fixed point.

3. Our final task is to show that any other fixed point of f - let's call it y - is actually above the top of the path we have constructed. So we assume that f(y) = y and from this prove that y is an upper bound of <u>every</u> point on the path we have constructed. From this it follows that y is as high as the <u>least</u> upper bound of the whole path. Again, we use mathematical induction.

> 3.1 To start with, the bottom point of the path is the bottom point of the whole domain. Everything is an upper bound of that, and so is y. That was easy.
>
> 3.2 We now assume that y is as high as some point on the path, and try to prove that it is as high as the next point too. Apply f both to the point on the path and to the fixed point y. Because f is monotonic, these two results are also ordered. The lower of them is the very next point on the path, whereas the higher of them is f(y). But y is by definition a fixed point of f, so f(y) equals y. So the next point of the path is also bounded above by y. Therefore, by induction, all points on the path have y as an upper bound. Since y was an arbitrary fixed point of f, we have proved that the least upper bound of the path is the least of all fixed points of f.

This third proof was perhaps the most difficult; it shows that among all the fixed points of f we have got the one we wanted, namely the one which is lowest in the specified scale of comparison.

Conclusion

Of course, Domain Theory has many other interesting theorems, which extend the usefulness of the fundamental theorem we have just proved. For

example, the cartesian product of two domains is a domain. This enables a pair of program units to be defined as the joint solution of a pair of mutually recursive equations. Even more remarkable, the space of all continuous functions from one domain to another (or to itself) is also a domain. This permits procedures with parameters to be defined by recursion. Finally, the concept of a domain can be slightly extended to ensure that the least upper bound of a chain of domains is also a domain. This allows data structures to be defined by recursion. The crowning achievement of Domain Theory, due to Dana Scott, is to show that domains also can be validly defined by recursion. This allows procedures to be passed freely as parameters or results of other procedures, and even of themselves. This is the theorem which lies at the basis of an untyped functional programming language like the lambda calculus. It forms the foundation of the type system for newer functional languages like ML or Miranda. It is the basis for the denotational semantic technique for formal definition of a wide variety of programming languages, including procedural ones. Such semantics are an essential prerequisite for the definition and proof of the correctness of the implementation of the language, and of every program written in the language. So it is of fundamental importance in Computing Science. Yet, until Dana Scott discovered it, mathematicians generally believed that such a domain could not exist. Now they are willing to accept it as a proof that even Computing Science exists.

Quite apart from its usefulness, Domain Theory is a wonderfully simple and elegant branch of mathematics, which is equally rewarding for study by pure mathematicians as by applied mathematicians, including Computer Scientists.

APPENDIX

This appendix introduces some of the traditional mathematical notations of domain theory, and uses them to give a formal proof of the fundamental theorem, namely that every continuous function on a domain has a fixed point, and one which is lower than every other fixed point in the domain ordering.

We will use the symbol $\sqsubseteq$ for the domain ordering; $x \sqsubseteq y$ means that x is equal to y or lower than y in the order $\sqsubseteq$. A relation $\sqsubseteq$ is called a partial order if it is reflexive, transitive, and antisymmetric. Each of these properties is defined by a formula as follows: in which x, y, and z range over all members of the domain.

(a) $\sqsubseteq$ is reflexive means

$$x \sqsubseteq x, \quad \text{for all } x$$

(b) $\sqsubseteq$ is transitive means

$$\text{if } x \sqsubseteq y \text{ and } y \sqsubseteq z \text{ then } x \sqsubseteq z, \text{ for all } x, y \text{ and } z$$

(c) $\sqsubseteq$ is antisymmetric means

$$\text{if } x \sqsubseteq y \text{ and } y \sqsubseteq x \text{ then } x = y, \text{ for all } x \text{ and } y.$$

In what follows we will assume that $\sqsubseteq$ is a partial order.

Theorem 0: for all x and y,

$$x = y \text{ exactly when } (x \sqsubseteq y \text{ and } y \sqsubseteq x)$$

Proof:
If $x = y$, then reflexivity of $\sqsubseteq$ (a) gives $x \sqsubseteq y$ and $y \sqsubseteq x$.
If $x \sqsubseteq y$ and $y \sqsubseteq x$, equality of x and y follows by

antisymmetry of $\sqsubseteq$ (c). So one of these assertions is true exactly when the other is.

Q. E. D.

Theorem 1: To prove that $x = y$, all you need is to prove

(*) $x \sqsubseteq z$ exactly when $y \sqsubseteq z$, for all z

Proof: Substitute x for z in the line marked by (*) to get

$$x \sqsubseteq x \text{ exactly when } y \sqsubseteq x .$$

But by reflexivity of $\sqsubseteq$, $x \sqsubseteq x$ is always true. It follows that

$$y \sqsubseteq x .$$

Similarly, substituting y for z in (*) we prove

$$x \sqsubseteq y .$$

Equality of x and y follows from the last two lines by antisymmetry of $\sqsubseteq$ (c).

Q. E. D.

A partial order of a domain has a bottom element satisfying the definition

(d) x is a bottom of $\sqsubseteq$ means

$$x \sqsubseteq y \text{ for all } y .$$

Theorem 2: A partial order has at most one bottom element.

Proof: suppose $x1$ and $x2$ are both bottom elements. Since

x1 is a bottom, we substitute it for x, and x2 for y in the definition of a bottom (d), getting

$$x1 \sqsubseteq x2$$

Similarly, since x2 is a bottom,

$$x2 \sqsubseteq x1.$$

Now the antisymmetry of $\sqsubseteq$ allows us to to conclude

$$x1 = x2$$

If any two things with some property can be proved identical, then there exists only one thing with that property.

Q. E. D.

In view of this theorem we will use a single symbol $\perp$ to denote the bottom of the partial order.

A point y is said to be the <u>least</u> of a set T if

y is in T, and $y \sqsubseteq z$ for all z in the set T

Theorem 3: No set can have more than one least member.

Proof: similar to the proof of uniqueness of $\perp$.

A point z is said to be an <u>upper bound</u> of a set S if

$x \sqsubseteq z$, for <u>all</u> x in the set S

Now let T be the set containing <u>all</u> upper bounds of S, and let y be the least member of T. Then y is said to be the <u>least upper bound</u> of S. Since there is at most one such point,

we introduce the notation lub(S) to denote it. Translating the definitions given above into symbols we get the defining law for least upper bounds:

(e) $lub(S) \sqsubseteq z$ means the same as

($x \sqsubseteq z$ for all x in S)

In the following theorems we will assume that lub(S) exists.

Theorem 4: $x \sqsubseteq lub(S)$, for all x in S.

Proof: Substituting lub(S) for z in the definition of lub (e):

$lub(S) \sqsubseteq lub(S)$ means the same as

($x \sqsubseteq lub(S)$ for all x in S)

But, by reflexivity of $\sqsubseteq$, $lub(S) \sqsubseteq lub(S)$ is always true. So the theorem, which has the same meaning, is also always true.

Q. E. D.

Let $\{w,z\}$ be the set containing just w and z. If $w = z$, it has only one member; otherwise it has exactly two.

Theorem 5: $lub(w,z) = z$ exactly when $w \sqsubseteq z$.

Proof: By theorem 0

$$lub(w,z) = z$$

exactly when

$$(lub(w,z) \sqsubseteq z \text{ and } z \sqsubseteq lub(w,z))$$

which holds exactly when

$$\text{lub}(w,z) \sqsubseteq z$$

because z is in (w,z), and theorem 4 tells us that

$$z \sqsubseteq \text{lub}(w,z) \quad \text{is always true.}$$

Now substitute (w,z) for S in the definition of lub (e):

$\text{lub}(w,z) \sqsubseteq z$ means the same as

($x \sqsubseteq z$ for all x in (w,z))

Since w and z are the only members of (w,z), this means the same as

$$w \sqsubseteq z \text{ and } z \sqsubseteq z,$$

which by reflexivity of $\sqsubseteq$ holds exactly when

$$w \sqsubseteq z,$$

So the theorem has been proved by a chain of equivalences leading from $\text{lub}(w,z)$ to $w \sqsubseteq z$.

Q. E. D.

Let $S \cup (\perp)$ be the set obtained by including $\perp$ into S.

Theorem 6: $\text{lub}(S \cup (\perp)) = \text{lub}(S)$

Proof: $x \sqsubseteq z$, for all x in $(S \cup (\perp))$

means the same as

$$((x \sqsubseteq z \text{ for all } x \text{ in } S) \text{ and } \bot \sqsubseteq z).$$

Since $\bot \sqsubseteq z$ for all z (d), this holds exactly when

$$x \sqsubseteq z, \text{ for all } x \text{ in } S$$

Using the definition of lub once on each of these equivalent assertions, we find that

$$\text{lub}(S \cup (\bot)) \sqsubseteq z .$$

holds exactly when

$$\text{lub}(S) \sqsubseteq z$$

Since we have proved this for any z, theorem 1 allows us to conclude

$$\text{lub}(S \cup (\bot)) = \text{lub}(S)$$

Q. E. D.

A set S is said to be a <u>chain</u> if

S is nonempty, and

for any pair x, y of its members, either $x \sqsubseteq y$ or $y \sqsubseteq x$.

Theorem 7: Every nonempty subset of a chain is a chain.

Proof: trivial.

Theorem 8: If $w \sqsubseteq z$ then the set (w,z) is a chain.

Proof: trivial.

The time has come to combine all the definitions we have given so far into a single definition, that of a domain. A set D is defined to be a domain if it has the following properties:

(1) its members are related by a partial order $\sqsubseteq$

(2) it contains a member $\perp$ which is the bottom of the partial order $\sqsubseteq$

(3) for every subset S of D, if S is a chain then the least upper bound lub(S) is a member of D (though not necessarily of S).

A function f from one domain to another (or to the same domain) is defined to be continuous if

$$f(\text{lub}(S)) = \text{lub}(\{ f(x) \mid x \in S \}) \text{ for all chains } S.$$

In future, we will talk only about a continuous function f.

Theorem 9: If $w \sqsubseteq z$ then $f(w) \sqsubseteq f(z)$

Proof: $w \sqsubseteq z$ by assumption

Therefore $\text{lub}(w,z) = z$ by theorem 5

Apply f to both sides of this equation:

$$f(\text{lub}(w,z)) = f(z).$$

By theorem 8, (w,z) is a chain and so by the assumption that f is continuous

$$f(\text{lub}(w,z)) = \text{lub}\{ f(x) \mid x \in (w,z) \}$$

$$= \text{lub}\{ f(w), f(z) \} \qquad \text{by set theory.}$$

By transitivity of equality,

$$\text{lub}\{ f(w),f(z) \} = f(z).$$

The desired conclusion follows by application of theorem 5 in the other direction.

Q. E. D.

A function from one domain to the same domain may be applied repeatedly. The result of applying it n times to an argument x is denoted $f^n(x)$. This is defined formally by induction on n :

$$f^0(x) = x, \qquad \text{for all } x$$

$$f^{n+1}(x) = f(f^n(x)), \qquad \text{for all } x \text{ and } n.$$

In future the letters n , m , and k will stand for natural numbers, i.e., non-negative integers.

Theorem 10: $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$ for all n .

Proof: by induction on n .

If $n = 0$, $f^0(\perp) = \perp$ by definition of f^0

$\perp \sqsubseteq f^1(\perp)$ by definition of $\perp$

Therefore $f^0(\perp) \sqsubseteq f^1(\perp)$ by substitution of equals.

Now assume $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$. By theorem 9 , a continuous function can be applied to both sides of this inequality, giving

$$f(f^n(\perp)) \sqsubseteq f(f^{n+1}(\perp)) .$$

Substitute the definition of $f^{n+1}(x)$ twice:

$$f^{n+1}(\perp) \sqsubseteq f^{(n+1)+1}(\perp).$$

This is a copy of the theorem, with $n+1$ replacing n ; so the theorem follows by induction.

Q. E. D.

Theorem 11: $f^m(\perp) \sqsubseteq f^{m+k}(\perp)$, for all m and k.

Proof: by induction on k.

If $k = 0$, $f^m(\perp) \sqsubseteq f^{m+0}(\perp)$ follows from the reflexivity of $\sqsubseteq$

Assume $f^m(\perp) \sqsubseteq f^{m+k}(\perp)$. By the previous theorem

$$f^{m+k}(\perp) \sqsubseteq f^{m+k+1}(\perp) .$$

From the last two lines by transitivity of $\sqsubseteq$

$$f^m(\perp) \sqsubseteq f^{m+k+1}(\perp).$$

Q. E. D.

Consider now a set S defined as the result of applying f any number of times to the bottom element of the domain.

$$S = \{ f^n(\perp) \mid n \leq 0 \}$$

$$= (\perp, f(\perp), f(f(\perp)), \ldots, f^n(\perp), \ldots).$$

It will be clearer to work with the second form of this definition in spite of the informal dots.

Theorem 12: S is a chain.

Proof: S is certainly non-empty, because it contains $\perp$. Now let x and y be members of set S. Then by definition of S, there are numbers m and n such that

$$x = f^m(\perp) \quad \text{and} \quad y = f^n(\perp).$$

Since m and n are numbers, either $m \leq n$ or $n \leq m$. In the first case, there is a k such that $m + k = n$, and so by the previous theorem

$$f^m(\perp) \sqsubseteq f^n(\perp)$$

i.e., $x \sqsubseteq y$.

In the second case, similar reasoning shows that $y \sqsubseteq x$. Since any pair of members is comparable, S satisfies the definition of a chain.

Q. E. D.

Since S is a chain, it has a least upper bound lub(S), which the following theorem shows to be a fixed point of S.

Theorem 13: $f(\text{lub}(S)) = \text{lub}(S)$

Proof: by the definition of S

$$f(\text{lub}(S)) = f(\text{lub}(\perp, f(\perp), f(f(\perp)), \ldots, f^n(\perp), \ldots)).$$

Because f is continuous, it distributes inside lub:

$$f(\text{lub}(S)) = \text{lub}\{f(\perp), f(f(\perp)), f(f(f(\perp))), \ldots f^{n+1}(\perp) \ldots\}.$$

By theorem 6, the insertion of $\perp$ into a set does not change its least upper bound, so

$$f(\text{lub}(S)) = \text{lub}\{\perp, f(\perp), f(f(\perp)), \ldots f^{n+1}(\perp), \ldots\}.$$

$= \text{lub}(S)$ by definition of S

The right hand side of the equation is the exact definition of S

Q. E. D.

Theorem 14: If $f(y) \sqsubseteq y$ then $f^n(\perp) \sqsubseteq y$.

Proof: by induction on n.

If $n = 0$, $f^0(\perp) = \perp \sqsubseteq y$, as proved before.

Assume $f^n(\perp) \sqsubseteq y$.

By theorem 9 we can apply f to both sides of an inequality, getting

$$f(f^n(\perp)) \sqsubseteq f(y).$$

By assumption, $f(y) \sqsubseteq y$, so transitivity of $\sqsubseteq$ gives

$$f(f^n(\perp)) \sqsubseteq y.$$

Applying the definition of f^{n+1} ,

$$f^{n+1}(\perp) \sqsubseteq y$$

That completes the induction step and the proof.

Q. E. D.

It remains only to prove that among all the fixed points of f lub(S) is the very least.

Theorem 15: If $f(y) = y$ then $lub(S) \sqsubseteq y$

Proof: by the assumption and reflexivity of $\sqsubseteq$

$$f(y) \sqsubseteq y .$$

By the previous theorem

$$f^n(\bot) \sqsubseteq y \quad \text{for all } n .$$

Since every member of S has the form $f^n(\bot)$, for some n :

$$x \sqsubseteq y \qquad \text{for all } x \text{ in } S$$

By one final glorious application of the definition of the least upper bound

$$lub(S) \sqsubseteq y .$$

Q. E. D.