Burt Rosenberg, July 30, 1996

0.0.1 Denotational Semantics

Denotational Semantics is a formalism for attaching meaning to programs. It seems quite circular. Many people involved in computers understand programs better than they do mathematics. Programs are more solid. The natural reaction is to understand mathematics in terms of programs, rather than programs in terms of mathematics. Programming becomes the formalism most people use to understand mathematics. Denotational Semantics uses mathematics, in particular, the lambda calculus and a thing called domain theory, to understand programs.

The largest first step is to consider seriously that a program might not halt. The halting problem states that it is not possible to decide ahead of time as to the safty of an arbitrary program. Hence, nondefihalting becomes a value, an alternative result returned by a function. It is denoted $\perp$, and is called *nontermination* or, for reasons to be explained below, *bottom.*

Here is a scheme program which "hits bottom":

```
> (define (N x)
      ( if (= x 0)
             (LOOP_FOREVER)  ;then
             x               ; else
       )
  )
> (define (LOOP_FOREVER)
      ((lambda(x)(x x))(lambda(x)(x x)))
  )
> (N 1)
1
> (N 0)

Interrupt: keyboard
1>
```

The complete mathematical description of this function which maps anything to itself except 0, which it maps to nontermination:

$$\begin{array}{rccl} N: & X & \rightarrow & X \\ & x & \mapsto & \left\{ \begin{array}{ll} x & \text{if } x \neq 0 \\ \perp & \text{else} \end{array} \right. \end{array}$$

This introduction of nontermination notation engenders even more notation. If a function returns nontermination, other functions may want to take nontermination as an input value. It is commonsense that a function should not terminate when given nontermination as its input value. Consider the behavoir of a program in which some called subroutine loops forever. The program also loops forever. Such functions are called *strict*, and the mathematically precise definition of strictness for a function f is that $f(\perp) = \perp$. The halting problem was an example of a nonstrict function: it mapped $\perp$ to false: $H(f(x))$ is false when $f(x) = \perp$, else $H(f(x))$ is true.

Any function $F : X \to X$ which has the possibility of non-halting, that is, it is not really a function on all of X but on $X' \subseteq X$ of values where $F(x)$ terminates, can be transformed into a function on the new richer domain

$$X_\perp = \{\, x \mid x \in X \text{ or } x = \perp \,\}$$

by defining $F(\perp) = \perp$ and $F(x) = \perp$ if $x \notin X'$. This is called the *lifted function*, and $X_\perp$ is the *lifted domain*:

$$\begin{array}{rccl} F_\perp : & X_\perp & \to & X_\perp \\ & x & \mapsto & \begin{cases} F(x) & \text{if } x \in X' \\ \perp & \text{else} \end{cases} \end{array}$$

Denotational Semantics handles the problem of nontermination by lifting all domains and functions consistent with strictness.

The second pure and somewhat daunting formalism employed by Denotational Semantics is the formalism of the *store.* A store is the model of computer memory, and in Denotational Semantics it is a map from locations to values. We denote the space of locations by $\mathcal{L}$. The value stored at a location can be fairly general, but let's assume that the store holds natural numbers $\mathcal{N}$. Hence a store S is a map $\mathcal{L} \to \mathcal{N}$. The space of stores $\mathcal{S}$ is *the space of all possible maps.* To update the store, that is, to change the value of a single location, is to transform one store into another:

$$\begin{array}{rccc} u : & \mathcal{L} \times \mathcal{N} \times \mathcal{S} & \to & \mathcal{S} \\ & (l, n, \sigma) & \mapsto & \sigma' \end{array}$$

where σ and σ' are functions related by:

$$\sigma'(l') = \begin{cases} \sigma(l') & \text{if } l' \neq l \\ n & \text{else.} \end{cases}$$

The processing of a program involves these sequence of updates to the store, with one store transforming into another.

A third formalism is that of Curried variables. A function on several variables can be reduced to a sequence of functions on each variable separately by using higher order functions, that is, functions which return functions. Consider a function $f : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$. Fixing the value of the first variable, say $x = a$, we have a function in a single variable y, $f(x, y) = f(a, y)$. We have this situation for each fixed a, so f can also be a function taking a natural, returning a function from naturals to naturals: $f : \mathcal{N} \to \mathcal{N} \to \mathcal{N}$. In this equivalence, the first variable is said to have been *Curried* into the function.

```
> (define f (lambda(x)
                (lambda(y) (+ x y))
          ))
> (define g (f 1) )
> (g 3)
4
>
```

Using this and the store formalism, we rewrite the update function $u : \mathcal{L} \to \mathcal{N} \to \mathcal{S} \to \mathcal{S}$,:

$$u = \lambda l.\lambda n.\lambda\sigma.\sigma[n/l],$$

that is, taking a store, a location and a value, update returns a new store, with the map $l \mapsto n$ substituted into the old store. You should not wonder how this substitution is done: Denotational Semantics is not about how, but what, is being denoted. Since σ exists, so can $\sigma[n/l]$, somehow accessed or created, out there is the space of all maps. Other store operations are access $s : \mathcal{L} \to \mathcal{S} \to \mathcal{N}$:

$$s = \lambda l.\lambda\sigma.\sigma(l),$$

and the creation of an empty store:

$$\nu = \lambda l.0$$

We are still a long way from a Denotional Semantics for anything respectible, but we shall go ahead with something despicable, hoping to introduce as early as possible the point of all the formalism. Denotational Semantics writes down a context free grammar for the language, a domain of values and operators for the semantics, and a mapping function to apply to syntax to reveal semantics.

THE SYNTAX:

```
P -> C                        ; program is commands
C -> E | E C | (set! L E) C ; commands are expressions or set!
E -> N | L | (+ E1 E2)      ; expressions are values, locations or sums
L -> a | b | ...             ; locations have names
N -> 0 | 1 | ...             ; values have numbers
```

THE SEMANTIC DOMAINS:

$$\begin{aligned}
\mathcal{N} &= \{0, 1, 2, \ldots\}, \\
& \quad + : \mathcal{N} \to \mathcal{N} \to \mathcal{N} \\
\mathcal{L} &= \{a, b, c, \ldots\} \\
\mathcal{S} &= \mathcal{L} \to \mathcal{N}, \\
& \quad u : \mathcal{L} \to \mathcal{N} \to \mathcal{S} \to \mathcal{S} \\
& \quad s : \mathcal{L} \to \mathcal{S} \to \mathcal{N} \\
& \quad \nu : \emptyset \to \mathcal{S}
\end{aligned}$$

THE SEMANTIC FUNCTIONS:

$$\begin{aligned}
\mathcal{P} &: \mathtt{P} \to \mathcal{N} \\
& \quad \mathcal{P}[[\mathtt{P}]] = s(a, \mathcal{C}[[\mathtt{C}]](\nu)) \\
\mathcal{C} &: \mathtt{C} \to \mathcal{S} \to \mathcal{S}
\end{aligned}$$

$$
\begin{array}{lll}
 & & \mathcal{C}[[\texttt{E}]](\sigma) = u(a, \mathcal{E}[[\texttt{E}]](\sigma), \sigma) \\
 & & \mathcal{C}[[\texttt{E C}]](\sigma) = \mathcal{C}[[\texttt{C}]](\mathcal{C}[[\texttt{E}]](\sigma)) \\
 & & \mathcal{C}[[\texttt{(set! L E) C}]](\sigma) = \mathcal{C}[[\texttt{C}]](u(\mathcal{L}[[\texttt{L}]], \mathcal{E}[[\texttt{E}]](\sigma), \sigma)) \\
\mathcal{E} & : & \texttt{E} \rightarrow \mathcal{S} \rightarrow \mathcal{N} \\
 & & \mathcal{E}[[\texttt{N}]](\sigma) = \mathcal{N}[[\texttt{N}]] \\
 & & \mathcal{E}[[\texttt{L}]](\sigma) = s(\mathcal{L}[[\texttt{L}]], \sigma) \\
 & & \mathcal{E}[[\texttt{(+ E1 E2)}]](\sigma) = \mathcal{E}[[\texttt{E1}]](\sigma) + \mathcal{E}[[\texttt{E2}]](\sigma) \\
\mathcal{L} & : & \texttt{L} \rightarrow \mathcal{L} \\
 & & \mathcal{L}[[\texttt{L}]] = L \\
\mathcal{N} & : & \texttt{N} \rightarrow \mathcal{N} \\
 & & \mathcal{N}[[\texttt{N}]] = N
\end{array}
$$

Double square brackets are traditional in the application of the semantic functions to the syntax elements (nonterminals of the grammar).

0.0.2 Denotational Semantics: Environments and Continuations

We now introduce identifiers into our semantics. We add grammar rules for the syntax of identifiers, a domain of values for the identifiers, and additional semantic functions to map the new grammar symbols according to their productions to "real" functions. Identifiers are bound to locations, the collection of all bindings is an *environment*. Any environment, then is a map from identifiers to locations. Composing this with a store, we have a map from identifiers to values.

THE SYNTAX:

```
C -> E | E C                              ; commands
E -> N | I | (E1 E2) | (lambda (I) E )    ; expressions (return values)
    | (lambda (I) C E) | (set! I E)
    | (if EC E0 E1)
I -> x | y | z | ...                      ; identifiers
N -> 0 | 1 | ...                          ; numbers
```

THE SEMANTIC DOMAINS:

$$
\begin{array}{rcl}
\mathcal{N} & = & \text{numbers} \\
 & = & \{0, 1, 2, \ldots\} \\
\mathcal{L} & = & \text{locations} \\
 & = & \{a, b, c, \ldots\}
\end{array}
$$

$$\begin{array}{rcl}
\mathcal{I} & = & \text{identifiers} \\
 & = & \{x, y, z, \ldots\} \\
\mathcal{U} & = & \text{environments} \\
 & = & \mathcal{I} \to \mathcal{L} \\
\mathcal{S} & = & \text{stores} \\
 & = & \mathcal{L} \to \mathcal{V} \\
\mathcal{V} & = & \text{values} \\
 & = & \mathcal{N} \ \amalg \ (\mathcal{E}, \mathcal{U}, \mathcal{L}) \\
\mathcal{E} & = & \text{continuations with value} \\
 & = & \mathcal{U} \times \mathcal{S} \to \mathcal{U} \times \mathcal{S} \times \mathcal{V} \\
\mathcal{C} & = & \text{continuations} \\
 & = & \mathcal{U} \times \mathcal{S} \to \mathcal{U} \times \mathcal{S}
\end{array}$$

THE SEMANTIC FUNCTIONS:

$$\begin{array}{rcl}
\mathcal{N} & : & \mathtt{N} \to \mathcal{N} \\
 & & \mathcal{N}[[\mathtt{N}]] = N \\
\mathcal{I} & : & \mathtt{I} \to \mathcal{I} \\
 & & \mathcal{I}[[\mathtt{I}]] = I \\
\mathcal{E} & : & \mathtt{E} \to \mathcal{E} \\
 & & \mathcal{E}[[\mathtt{N}]] = \lambda u.\lambda\sigma.\langle\, u, \sigma, \mathcal{N}[[\mathtt{N}]]\,\rangle \\
 & & \mathcal{E}[[\mathtt{I}]] = \lambda u.\lambda\sigma.\langle\, u, \sigma, \sigma \circ u(\mathcal{I}[[\mathtt{I}]])\,\rangle \\
 & & \mathcal{E}[[\mathtt{(E1\ E2)}]] = \lambda u.\lambda\sigma.((\lambda\epsilon_1.\lambda\epsilon_2.R)(\mathcal{E}[[\mathtt{E1}]]u\sigma)(\mathcal{E}[[\mathtt{E2}]]u\sigma)) \\
 & & \quad R = (\lambda\epsilon.\langle\, u, \pi_2\epsilon, \pi_3\epsilon\,\rangle)((\pi_1\pi_3\epsilon_1)(\pi_2\pi_3\epsilon_1)\{[\pi_3\pi_3\epsilon_1 \to \pi_3\epsilon_2]\sigma\}) \\
 & & \mathcal{E}[[\mathtt{(lambda\ (I)\ E)}]] = \lambda u.\lambda\sigma.\langle\, u, \sigma, (\lambda l.P)\nu(\sigma)\,\rangle \\
 & & \quad P = \langle\, \mathcal{E}[[\mathtt{E}]], \{[\mathcal{I}[[\mathtt{I}]] \to l]u\}, l\,\rangle \\
 & & \mathcal{E}[[\mathtt{(lambda\ (I)\ C\ E)}]] = \lambda u.\lambda\sigma.\langle\, u, \sigma, (\lambda l.P')\nu(\sigma)\,\rangle \\
 & & \quad P' = \langle\, \mathcal{E}[[\mathtt{E}]] \circ \mathcal{C}[[\mathtt{C}]], \{[\mathcal{I}[[\mathtt{I}]] \to l]u\}, l\,\rangle \\
 & & \mathcal{E}[[\mathtt{(set!\ \ I\ E)}]] = \lambda u.\lambda\sigma.(\lambda\epsilon.\lambda l.\ \langle\, u, \{[l \to \pi_3\epsilon]\sigma\}, \pi_3\epsilon\,\rangle)(\mathcal{E}[[\mathtt{E}]]u\sigma)(u(\mathcal{I}[[I]])) \\
 & & \mathcal{E}[[\mathtt{(if\ EC\ E0\ E1)}]] = \lambda u.\lambda\sigma.\big((\lambda n.n \to \mathcal{E}[[\mathtt{E0}]] \,\square\, \mathcal{E}[[\mathtt{E1}]])\pi_3(\mathcal{E}[[\mathtt{EC}]]u\sigma)\big)u\sigma \\
\mathcal{C} & : & \mathtt{C} \to \mathcal{C} \\
 & & \mathcal{C}[[\mathtt{E}]] = \lambda u.\lambda\sigma.(\lambda\epsilon.\langle\, \pi_1\epsilon, \pi_2\epsilon\,\rangle)(\mathcal{E}[[\mathtt{E}]]u\sigma) \\
 & & \mathcal{C}[[\mathtt{E\ C}]] = \lambda u.\lambda\sigma.(\mathcal{C}[[\mathtt{C}]])((\lambda\epsilon.\langle\, \pi_1\epsilon, \pi_2\epsilon\,\rangle)(\mathcal{E}[[\mathtt{E}]]u\sigma))
\end{array}$$

Double square brackets are traditional in the application of the semantic functions to the syntax elements (nonterminals of the grammar). The additional functions are defined:

$$\begin{array}{rcl}
\nu(\sigma) & : & \mathcal{S} \to \mathcal{L} \\
 & & s \mapsto \text{ any unused } l \in \mathcal{L} \\
\cdot \to \cdot \,\Box\, \cdot & : & \mathcal{N} \times \mathcal{E} \times \mathcal{E} \to \mathcal{E} \\
 & & (n, \epsilon_1, \epsilon_2) \mapsto \begin{cases} \epsilon_1 & \text{if } n = 0 \\ \epsilon_2 & \text{if } n \neq 0 \end{cases} \\
\{[\,\cdot \to \cdot\,]\ \cdot\} & : & \mathcal{L} \times \mathcal{V} \times \mathcal{S} \to \mathcal{S} \\
 & & (l, v, \sigma) \mapsto \{\, (l', v') \in \sigma \mid l' \neq l \,\} \cup \{\, (l, v) \,\} \\
\{[\,\cdot \to \cdot\,]\ \cdot\} & : & \mathcal{I} \times \mathcal{L} \times \mathcal{U} \to \mathcal{U} \\
 & & (i, l, u) \mapsto \{\, (i', l') \in u \mid i' \neq i \,\} \cup \{\, (i, l) \,\}
\end{array}$$

0.0.3 Denotational Semantics: Fixpoints and Recursion

We complete our tour of Denotational Semantics with a discussion of the meaning of recursion. Consider a while loop. Under the intuitive explanation of Operational Semantics, the meaning of a while loop is a repetition of the body of the loop, n times, where n is a nonnegative integer. Denotational Semantics requires a wider perspective. Given the function which denotes the body of the while loop, we need to derive a new function which summarizes the action of the body repeated as required by the condition of the while loop.

Let us look at recursion for an example of the difficulties involved. Suppose we have this recursive definition:

$$F(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ F(x+1) & \text{else} \end{cases}$$

Here is a possible function F which satisfies this recursive definition:

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ \perp & \text{else} \end{cases}$$

Here is another:

$$f'(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{else} \end{cases}$$

The meaning of the function can be either of these alternatives. If the meaning is assigned by Operational Semantics, and we take the recursive definition of F literally, call $F(x+1)$ when given a positive x, then the first alternative is taken. Denotational Semantics will also chose this alternative, without requiring that the non-termination be simulated on a model machine.

The approach is to consider a recursive function as a limit of a series of approximations to the function. Each approximation limits the recursion to less than a fixed depth. As the depth limit is pushed to infinity, a sequence of approximating functions converges to a limit function. The limit function is taken as the "value" of the approximating series. The concept of a series of values approaching a limit requires two more basic notions: distance and continuity. Distance is required in order to know what it means for one thing to approach another. Continuity is needed so that the final value after completing the infinite sequence of approximations is as expected, that it doesn't somehow jump away at the last minute from the value to which the sequence tends.

The notion of distance can be rephrased as a notion of order.

Definition 1 *A Partial Order is a relation $\geq$ on a set P which satisfies three axioms:*

- *Reflexivity: for all $p \in P$, $p \geq p$.*
- *Transitivity: for $p, q, r \in P$, if $p \geq q$ and $q \geq r$ then $p \geq r$.*
- *Antisymmetry: for any $p, q \in P$, if $p \geq q$ and $q \geq p$, the $p = q$, that is, they are the same element of P.*

A set with a partial order is called a poset.

An example of a poset is intervals on the real line ordered by point inclusion:

$$[a, b] \geq [c, d] \text{ if } [a, b] \subseteq [c, d]$$

Certain schools of logic, topology and computer science look at this intuitively by saying that the interval $[a, b]$ contains more information than the interval $[c, d]$, because what is trying to be indicated is a certain undiscovered x inside the interval. Perfect information is the one point interval $[a, a]$, complete lack of information is the entire real line $[-\infty, \infty]$. This state of complete ignorance is called "bottom" and denoted $\perp = [-\infty, \infty]$. It is always true that $[a, b] \geq \perp$ for any interval $[a, b]$.

In the case of calculating some real number x, we start at $\perp$, only knowing that it is a number that we are trying to calculate, and we work our way through successive approximations, gotten through finite means (finite calculations on a machine of finite resources) towards, in the limit, and perhaps *only* in the limit, perfect information about x:

$$\perp \leq [a_1, b_1] \leq [a_2, b_2] \leq \ldots \leq [x, x].$$

Note that $p \leq q$ is defined for the poset to be another way of writing $q \geq p$.

We have reconsidered distance as an order, shorter things versus longer things. We now consider the abstraction of continuity, which considers how functions on posets respond to infinite sequences of order related elements.

Definition 2 *Let s be a set of elements in a poset P, $s \subseteq P$. An upper bound of s is an element p of P such that $s' \leq p$ for all $s' \in s$. The least upper bound of s is an upper bound p of s such that any other upper bound p' is at least as large $p \leq p'$.*

Whether or not a set has an upper bound and a least upper bound depends upon the structure of the poset.

Definition 3 *A domain is a poset P such that there is a least element $\perp$, that is, $\perp \leq p$ for any $p \in P$, and that any chain:*

$$p_1 \leq p_2 \leq \ldots$$

has a least upper bound, denoted $\vee_i p_i$.

Denotational Semantics restricts its attention to domains. Data structures as well as the target value domains for semantics, including treating functions as values, are all required (or fixed-up) to be domains.

Definition 4 *Let P and Q be domains and a function $f : P \to Q$. The function f is monotonic if for any $p, p' \in P$ such that $p \leq p'$, then $f(p) \leq f(p')$. A monotonic function f is said to be continuous if for all chains $p_1 \leq p_2 \leq \ldots$, the function "commutes with least upper bounds":*

$$f(\vee_i p_i) = \vee_i f(p_i).$$

Denotational Semantics restricts its attention to continuous functions of domains. Recursion equations give continuous functions.

There are also more general arguments for this restriction. We adopted a viewpoint where elements of the poset $x_1, x_2, \ldots$ converge upon the exact answer x, where each x_i has the disadvantage of being only an approximation to x, but the advantage of being calculated with only finite resource. As resources are increased, our knowledge increases, $x_i \leq x_j$ for $i \leq j$, hence we have a chain, whose least upper bound is the true solution, $x = \vee_i x_i$. We now use x or its approximates as input to another function f. Each approximate of x gives us an aproximate of $f(x)$. If f is computable with finite resource, than what we can possibly know about $f(x)$ is the sequence of approximations $f(x_1), f(x_2), \ldots$. This sequence tends to $\vee_i f(x_i)$. The method of approximations is of use to us only if this "implied" limiting value matches the true value of $f(x)$, that is, only if:

$$\vee_i f(x_i) = f(\vee_i x_i).$$

By requiring continuity on the domain functions, we assure that a function of a sequence of approximations indicates correctly the value of the function on the limit of the approximations.

We now construct the domain for functions from naturals to naturals.

Definition 5 *Let $\mathcal{N}_\perp$ be the set of naturals and the new element $\perp$. Impose the order:*

$$n \leq m \textit{ if and only if } n = m \textit{ or } n = \perp.$$

This gives a domain. (Domains such as these, with a smallest element an all other elements larger and incomparable, is called a flat domain.) Given two functions f and g on the flat domain $\mathcal{N}_\perp$, impose the order,

$$f \leq g \textit{ if and only if } \forall n \in \mathcal{N}_\perp,\ f(n) \leq g(n).$$

This turns the set of all functions $\mathcal{N}_\perp \to \mathcal{N}_\perp$ into a domain, with smallest element $\perp$, the everywhere undefined function:

$$\perp(n) = \perp, \textit{ for all } n \in \mathcal{N}_\perp.$$

We stated that the posets just defined are domains, but we must backup this assertion with a proof. In particular, we must show that any chain has a least upper bound.

Theorem 1 *Let $\{f_1, f_2, \ldots\}$ be a chain of functions in the domain of functions $\mathcal{N}_\perp \to \mathcal{N}_\perp$, that is, $f_i \leq f_j$ for any $i \leq j$. The least upper bound of the chain is:*

$$(\vee_i f_i)(n) = \begin{cases} f_j(n) & \text{if } f_j(n) \neq \perp \text{ for some } j \\ \perp & \text{if } f_i(n) = \perp \text{ for every } i \end{cases}$$

PROOF: Left to the reader.

Given two function $f, g : \mathcal{N}_\perp \to \mathcal{N}_\perp$, we have that $f \leq g$ if g contains more information than f. If $f(n) = m$, then it must be that $g(n) = m$, so g contains at least as much information as f. However $g(n)$ might yield a value whereas $f(n) = \perp$. The function g gives data where f is undecided. A continuous function on the domain $\mathcal{N}_\perp \to \mathcal{N}_\perp$ is a map from functions to functions:

$$F : (\mathcal{N}_\perp \to \mathcal{N}_\perp) \to (\mathcal{N}_\perp \to \mathcal{N}_\perp)$$

such that,

1. If $f \leq g$, then $F(f) \leq F(g)$.
2. The function F commutes with limits: if $f_1 \leq f_2 \leq \ldots$ is any chain of functions, $F(\vee f_i) = \vee_i F(f_i)$.

It gets confusing to give the same name to functions and functions of functions, so a function of functions is sometimes called a *functional*. We say then that F is a continuous functional.

Continuous functionals are the natural result of recursive definitions. We provide an example. Given a function f, we transform f to $F(f)$:

$$F(f)(n) \quad = \quad \begin{cases} 1 & \text{if } n = 0 \\ nf(n-1) & \text{else} \end{cases}$$

We use the convention that multiplication is strict, n times $\perp$ is $\perp$. This is a recursive definition for the factorial. What that means is that factorial is a function which, when plugged in for f, gives you the factorial back again: $F(!) =!$, where ! is the factorial function. A solution to the functional equation $F(f) = f$ is called a *fixed point* of F. We saw that recursive equations for functions give functional equations whose fixed points are functions which fit the recursive descrption. There maybe more than one fixed point for $F(f) = f$, as we saw that there may be more than one function which fits a recursive description. We eliminate all but one by assigning as the meaning of the description the *least fixed point* of $F(f)$.

Theorem 2 (The least fixed point theorem) *Let F be a continuous functional on domain $\mathcal{N}_\perp \to \mathcal{N}_\perp$. Then there exists a least fixed point to the equation $F(f) = f$. It is given by the least upper bound of the chain:*

$$\perp \leq F(\perp) \leq F(F(\perp)) \leq \ldots$$

PROOF: Let $f_0 = \perp$ and $f_i = F(f_{i-1})$. Since $\perp \leq f$ for any function f, then $f_0 \leq f_1$. By the property of F, $f_1 = F(f_0) \leq F(f_1) = f_2$, hence,

$$f_0 \leq f_1 \leq f_2 \leq \ldots$$

Since $\mathcal{N}_\perp \to \mathcal{N}_\perp$ is a domain, every chain has a least upper bound. Hence $f = \vee_i f_i$ exists.

If g is a fixed point of F, then $g \geq f$. It is alwas true that $g \geq \perp$. Take F of both sides, $F(g) \geq F(\perp) = f_1$. But $F(g) = g$. So $g \geq f_1$. Continuing, $g \geq f_i$ hence g is greater than the least upper bound of the f_i.

Finally, we need to show that f itself is a fixed point. Since F is continuous:

$$F(f) = F(\vee_i f_i) = \vee_i F(f_i) = \vee_i f_i = f.$$

EXAMPLE: The Denotational Semantics of a while loop. Given the store σ, a while loop is of the form `while` $(\mathcal{C}\sigma)$ $\{\mathcal{S}\sigma\}$. We can develop a domain for stores, or consider the entire computer memory to be just one big number. Then the meaning of the while loop is the least fixed point of $F(f) = f$ where:

$$F(f)\sigma \quad = \quad \begin{cases} f \circ \mathcal{S}\sigma & \text{if } \mathcal{C}\sigma \\ \sigma & \text{else} \end{cases}$$

Note that the least fixed point theorem gives the solution for f explicitly as the least upper bound of:

$$\perp \leq \sigma \text{ if } \mathcal{C}\sigma \text{ else } \perp \leq \sigma \text{ if } \mathcal{C}\sigma \text{ else } \mathcal{S}\sigma \text{ if } \mathcal{C}\mathcal{S}\sigma \text{ else } \perp \leq \ldots$$