THE WHILE LOOP PROBLEM

None of our languages have had a loop construct. The most we have had is a *diverge* command that stands in for a non-terminating form that we handled using lifted domains. It is time to bring the loop into the picture and see if we can handle it in a proper denotational way. What is this way? One of our goals all along has been to separate syntax and semantics completely in the valuation functions, and to make the denotations serve as static meanings of the syntax. In other words, we want to attach semantic objects to syntax, and those objects can have no reference to anything else except more semantics.

If we attempt to define the while loop using a valuation function using the simple Store domain, we get the following, which is clearly operational in flavor:

\[
\text{while } \text{B do } \text{C} = \lambda s. \text{B}[\text{B}[s]] \rightarrow \text{C[while } \text{B do } \text{C]}(\text{C}[s]) \uplus s
\]

This is acceptable if all we want to do is to derive the meanings of programs with loops that terminate. We can unfold the recursion just as we did in the operational rules for the while loop and end up with a satisfactory result. However, this function does not follow the major constraint we placed on ourselves – that of providing a static semantic object as the meaning. It also fails to provide a meaning for a loop that does not terminate. If we look at the function in another light, we see that it is actually an equation with one unknown – the true meaning of the loop. In fact we have the same problem for any recursive definition. Take, for example, the following function:

\[
\text{q } = \lambda n. n \text{ equals zero } \rightarrow \text{one } \uplus \text{q (n plus one)}
\]

A quick glance reveals that this function only returns an answer for zero (assuming n must be a natural number) and does not terminate otherwise. So what is the meaning of q? Clearly we could give a solution that matches this description:

\[
q = \lambda n. n \text{ equals zero } \rightarrow \text{one } \uplus \bot
\]

However, this is not the only solution. Unfortunately, there are an infinite number of them. The requirements for any solution are clear:

\[
\begin{align*}
q(\text{zero}) &= \text{one} \\
q(\text{one}) &= q(\text{two}) \\
q(\text{two}) &= q(\text{three}) \\
& \vdots
\end{align*}
\]

So any function that produces the same value for the inputs greater than zero will work. For instance:

\[
q = \lambda n. n \text{ equals zero } \rightarrow \text{one } \uplus \text{four}, \text{ or }
\]

\[
q = \lambda n. n \text{ equals zero } \rightarrow \text{one } \uplus \text{twoHundred}
\]

are both solutions as well. Is there a solution to this, or are we forced to abandon our search for ‘the’ answer to a recursive equation? The answer is that if we relax our conditions a little, we can indeed find a unique answer to any recursive equation. Our conditions, restated are:

1. There is at least one solution.
2. We can choose the ‘best’ solution, whatever that means, if there are more than one.
3. The solution should match our operational intuition as closely as possible.

In order to investigate how this is possible, we will look at a real function, although our method should also work for the function q, as it should for any recursive equation.

THE FACTORIAL FUNCTION

The factorial function will serve as an example of a recursive function whose solution we already know. The function is:
\[
\lambda \cdot \text{fac } n \cdot n \text{ equals zero } \rightarrow \text{ one } \square n \text{ times } (\text{ fac } (n \text{ minus one}))
\]

Let us write the solution to this equation as \( \text{ fac } = \text{ factorial } \). Another way to say this is since \( \text{ factorial}(i) = i! \), then the solution is \( \text{ fac}(i) = i! \). Our job is show that \( \text{ factorial } \) is the best solution to the equation. We start by considering the unfoldings of the equation. If we substitute \( n \text{ minus one } \) for \( n \) in the recursion, we get:

\[
\lambda \cdot \text{fac } n \cdot n \text{ equals zero } \rightarrow \text{ one } \\
\square n \text{ times } ((n \text{ minus one}) \text{ equals zero } \rightarrow \text{ one } \\
\square ((n \text{ minus one}) \text{ times } (\text{ fac } ((n \text{ minus one}) \text{ minus one})))
\]

and then

\[
\lambda \cdot \text{fac } n \cdot n \text{ equals zero } \rightarrow \text{ one } \\
\square n \text{ times } ((n \text{ minus one}) \text{ equals zero } \rightarrow \text{ one } \\
\square ((n \text{ minus one}) \text{ times } (\text{ fac } ((n \text{ minus one}) \text{ minus one}) \text{ minus one})))
\]

We can do these substitutions for ever an produce a whole family of functions. In fact, if we disallow the recursion to take place, we can still produce \( \text{ factorial } \) as follows:

Let \( \text{ fac}_i \) be the function with \( i \) unfoldings. i.e.

\[
\text{ fac}_i = \lambda \cdot n \cdot n \text{ equals zero } \rightarrow \text{ one } \square n \text{ times } (\text{ fac } (n \text{ minus one}))
\]

\[
\text{ fac}_2 = \lambda \cdot n \cdot n \text{ equals zero } \rightarrow \text{ one } \\
\square n \text{ times } ((n \text{ minus one}) \text{ equals zero } \rightarrow \text{ one } \\
\square ((n \text{ minus one}) \text{ times } (\text{ fac } ((n \text{ minus one}) \text{ minus one})))
\]

e tc.

What can these functions do without invoking the recursion. If we write out the graph (the extension) of each function we get the following (by writing the derivations):

\[
\text{ graph } (\text{ fac}_i) = \{[\text{ zero}, \text{ one}]\}
\]

\[
\text{ graph } (\text{ fac}_2) = \{[\text{ zero}, \text{ one}],[\text{ one}, \text{ one}]\}
\]

\[
\text{ graph } (\text{ fac}_3) = \{[\text{ zero}, \text{ one}],[\text{ one}, \text{ one}],[\text{ two}, \text{ two}]\}
\]

\[
\text{ graph } (\text{ fac}_4) = \{[\text{ zero}, \text{ one}],[\text{ one}, \text{ one}],[\text{ two}, \text{ two}],[\text{ three}, \text{ six}]\}
\]
e tc.

The pattern here is that

\[
\text{ graph } (\text{ fac}_{i+1}) = \{[\text{ zero}, \text{ one}],[\text{ one}, \text{ one}],[\text{ two}, \text{ two}],[\text{ three}, \text{ six}]\}[i, i!]
\]

Clearly \( \text{ graph}(\text{ fac}_i) \subseteq \text{ graph}(\text{ fac}_{i+1}) \) for all \( i > 0 \)

In fact since we know what \( \text{ factorial } \) is, we can also state that

\[
\text{ graph}(\text{ fac}_i) \subseteq \text{ graph}(\text{ factorial}), \text{ and then}
\]

\[
\bigcup_{i=0}^{\infty} \text{ graph}(\text{ fac}_i) \subseteq \text{ graph}(\text{ factorial}), \text{ since each graph is contained in the next one.}
\]

Reasoning the other way, consider a member \( [a, b] \) of \( \text{ factorial } \). This arbitrary member must then be a member of a \( \text{ graph}(\text{ fac}_i) \) for some \( i \). This implies that
\[
\text{graph(factorial)} \subseteq \bigcup_{i=0}^{\infty} \text{graph}(\text{fac}_i)
\]

And so, from both results

\[
\text{graph(factorial)} = \bigcup_{i=0}^{\infty} \text{graph}(\text{fac}_i)
\]

Our strategy is thus clear. We will look for solutions to recursive equations by looking all of the recursive unfoldings of the recursion, and adding each successive solution to the set. The grand union of all of these will be the answer we are looking for; the solution will be expressed in terms of an infinite family of non-recursive functions, each one produced by one more unfolding from the last. It remains to figure out what \( \text{fac}_0 \) is. Without any unfolding, the function cannot operate at all, so its graph is empty. i.e.

\[
\text{graph}(\text{fac}_0) = \emptyset
\]

We can utilize bottom to good effect here, and write

\[
\text{fac}_0 = \lambda n. \bot, \text{ since we are using bottom to represent non-termination, i.e. a function that produces no results.}
\]

**FIXED POINT SOLUTIONS**

The next step is to generalize this result to be applicable to any recursive function. First we parameterize our equations for \text{factorial}. We will define

\[
F = \lambda f. \lambda n. n \text{ equals zero } \rightarrow \text{one } \sqcup n \text{ times } (f (n \text{ minus one}))
\]

which is a non-recursive function. Then

\[
\text{fac}_{i+1} = F (\text{fac}_i).
\]

By expanding these successively, we get

\[
\text{fac}_1 = F (\text{fac}_0)
\]

\[
\text{fac}_2 = F (\text{fac}_1) = F (F (\text{fac}_0))
\]

\[
\vdots
\]

\[
\text{fac}_i = F (\cdots (F (\text{fac}_0))) \text{, } i \text{ times}
\]

Therefore if we write the union of the graphs of each side, we get

\[
\bigcup_{i=0}^{\infty} \text{graph}(\text{fac}_i) = \bigcup_{i=0}^{\infty} \text{graph}(F^i (\emptyset))
\]

where \( F^i = F \circ F \circ F \cdots \), and \( \emptyset = (\lambda n. \bot) \)

The left hand side is just \( \text{graph(factorial)} \) that we derived before. So we have

\[
\text{graph(factorial)} = \bigcup_{i=0}^{\infty} \text{graph}(F^i (\emptyset))
\]

Also we can write \( \text{graph}(F (\text{factorial})) = \text{graph}(\text{factorial}) \), since applying \( F \) to \text{factorial} gives what we know to be the factorial function. Thus \text{factorial} is a fixed point of \( F \), since when \( F \) is a applied to it, we get the same thing back. That is how we got \( F \), after all. We have almost everything in place for the solution we are looking for, but we need to generalize from graphs of functions, i.e. sets to functions themselves. Before we do this, let us briefly return to the function \( q \), we started with. Let us write

\[
Q = \lambda g. \lambda n. n \text{ equals zero } \rightarrow \text{one } \sqcup g (n \text{ plus one})
\]

Then
\[ Q^0 = (\lambda n. \bot) \]
\[ \text{graph}(Q^0(\emptyset)) = \{\} \]
\[ Q^1(\emptyset) = \lambda n. n \text{ equals zero } \rightarrow \text{one } \bot (\lambda n. \bot)(n \text{ plus one}) \]
\[ = \lambda n. n \text{ equals zero } \rightarrow \text{one } \bot \]
\[ \text{graph}(Q^1(\emptyset)) = \{[\text{zero}, \text{one}]\} \]
\[ Q^2 = Q(Q(\emptyset)) \]
\[ = \lambda n. n \text{ equals zero } \rightarrow \text{one } \bot (n \text{ plus one } \rightarrow \text{one } \bot) \]
\[ \text{graph}(Q^2(\emptyset)) = \{[\text{zero}, \text{one}]\} \]

All the rest are the same, since all other arguments map to bottom (which can be omitted from the graph). Thus we have

\[ \bigcup_{i=0}^{\infty} \text{graph}(Q^i(\emptyset)) = \{[\text{zero}, \text{one}]\} \]

Since we formed \( Q \) just like we formed \( F \), we can say that the function that has this graph (call it \( qlimit \)) is a fixed point of \( Q \), i.e. \( Q(qlimit) = qlimit \). From our discussion before, we know that \( qlimit \) is not a unique solution; in fact, there are an infinite number of them. Call these functions \( qk \). We can also say that

\[ Q(qk) = qk \], i.e. they are all fixed points of \( Q \), and

\[ \text{graph}(qlimit) \subseteq \text{graph}(qk) \], i.e. the graph of \( qlimit \) is at least as small as the graph of any solution.

We say that \( qlimit \) is the least fixed point solution; this is the ‘best’ solution we were looking for. The method is now complete. In order to solve a recursive equation to find the best solution, we find the least fixed point of the equation

\[ f = F(f) \]

where \( f \) is any recursively defined function and \( F \) is the functional formed by parameterizing the function. The graph of the least fixed point solution is given by

\[ \bigcup_{i=0}^{\infty} \text{graph}(F^i(\emptyset)) \]

After examining the method more formally, we will write \( \text{fix } F \) as the this solution where \( \text{fix} \) is the least fixed point operator.

**GENERALIZING THE METHOD**

There are three parts to the generalization:

1. the graph subset is generalized to partial order on functions
2. constraining functions to be continuous
3. showing that domains are continuous partial orders

We are not going to study the whole formulation, but just its results, and try to motivate these results through simple examples.

A partial order is a relation on a domain, \( D \), that is reflexive, antisymmetric, and transitive. We will write \( a \preceq b \), where \( a, b \in D \)
and read it as “\( a \) is less defined than \( b \)”.
If we take the power set of a set \( E \), we can replace \( \leq \) by \( \subseteq \) and then the power set is a partially ordered set. However, we don’t need subset. For instance the set \( \{ a, b, c, d \} \) can be a partially ordered set if, for instance
\[
d \leq b, \text{ and } b \leq c
\]
Here \( a \) is unrelated to the others, but that is OK; there is no requirement for all elements to take part in the relationship. In the same way the subsets \( \{ a, b \} \), and \( \{ b, c \} \) are unrelated in the partial ordering on the powerset of \( \{ a, b, c \} \). The corresponding form to the empty set, which is a subset of all sets, in a partial ordering is bottom, i.e. \( \bot \).

For a partial ordering on domain \( D \), if there is an element \( b \in D \), such that \( \forall d \in D, b \leq d \), then \( b \) is the least element, and will be written as \( \bot \).

We now define the least upper bound (LUB) and greatest lower bound (GLB) in order to get to the point where we can discuss our chains of functions \( F^i(\emptyset) \).

The LUB for a domain \( D \) of a subset \( X \subseteq D \) is defined by
\[
\forall x \in X, x \leq \bigcup X, \text{ and } \forall d \in D \forall x \in X, x \leq d \Rightarrow \bigcup X \leq d
\]
Essentially this says that the LUB (if it exists) is the most defined element of the set. The GLB can be similarly defined. If bottom is in the domain, then \( \bigcap D = \bot \). There can also be a corresponding most-defined element, called top, written as \( \bigcap D = \top \).

**CHAINS AND CONTINUOUS FUNCTIONS**

A chain represents a family of elements in a domain that are consistent one with another. They are defined by

For a partially ordered domain \( D \), a subset \( X \) of \( D \) is a chain if and only if \( X \neq \emptyset \wedge \forall a, b \in X, a \leq b \lor b \leq a \).

Since we will need our chains to have least upper bounds, we need to restrict the kinds of partial orders we will accept. The complete partial order (CPO) is one in which all chains in the domain have a LUB. Further, it is a pointed CPO if it has a least element (bottom).

Two conditions will lead us to this point when we consider domains of functions. The first is monotonicity:

A function \( f : A \to B \) is monotonic if and only if \( \forall x, y \in A, x \leq y \Rightarrow f(x) \leq f(y) \)

The vast majority of the normal non-recursive or recursively defined functions are monotonic. They process their arguments in consistent ways.

The second is continuity:

For complete partial orders \( A \) and \( B \), a monotonic function \( f : A \to B \) is continuous if and only if for any chain \( X \subseteq A \), \( f(\bigcup X) = \bigcup \{ f(x) \mid x \in X \} \).

Thus continuous functions preserve the chains that are made up from the elements of the domain. If the elements form a chain, then so will the results of applying the function to those elements.

**LEAST FIXED POINT SOLUTIONS**

Finally, we can express the semantics of recursive functions in the terms we have laid out for ourselves. Firstly, we can state that functionals, such as the one derived from factorial are continuous functions. We can then state

For a functional \( F : D \to D \) and an element \( d \in D \), \( d \) is a fixed point of \( F \) if and only if \( F(d) = d \). \( d \) is the least fixed point of \( F \), if \( \forall e \in D, F(e) = e \Rightarrow d \leq e \).

The most important theorem is then the least fixed point theorem:
If the domain $D$ is a pointed CPO, the least fixed point of a continuous functional $F : D \to D$ exists and is defined to be $\text{fix } F = \bigsqcup \{ F^i(\bot) \mid i \geq 0 \}$, where $F^i = F \circ F \circ \cdots F, i \text{ times}$.

To prove this, first show $\text{fix } F$ is a fixed point of $F$.

$$F \left( \text{fix } F \right) = F \left( \bigsqcup \{ F^i(\bot) \mid i \geq 0 \} \right) = \bigsqcup \{ F \left( F^i(\bot) \right) \mid i \geq 0 \}, \text{ by continuity of } F$$

$$= \bigsqcup \{ F^i(\bot), F^2(\bot), F^3(\bot), \ldots \}$$

$$= \bigsqcup \{ F^i(\bot) \mid i \geq 1 \}$$

$$= \bigsqcup \{ \{ F^0(\bot) \} \cup \{ F^i(\bot) \mid i \geq 1 \} \}, \text{ since } F^0(\bot) = \bot, \text{ and } \forall i \geq 0, \bot \sqsubseteq F^i(\bot)$$

$$= \bigsqcup \{ F^i(\bot) \mid i \geq 0 \}$$

$$= \text{fix } F$$

So the meaning of a recursive definition $f = F(f)$ is $\text{fix } F$, the least fixed point functional denoted by $F$.

To show that it is the least fixed point, let $e \in D$ be a fixed point of $F$. Since $\bot \sqsubseteq e$, and, since $F$ is monotonic, $F^i(\bot) \sqsubseteq F^i(e) = e$. This implies that $\text{fix } F = \bigsqcup \{ F^i(\bot) \mid i \geq 0 \} \sqsubseteq e$, so $\text{fix } F$ must be the least fixed point.

**FACTORIAL AGAIN**

Going back to factorial:

$\text{fac} = \lambda n. n \text{ equals zero } \rightarrow \text{ one } \sqcup n \text{ times } \left( \text{fac} \left( n \text{ minus one} \right) \right)$

The functionality is $\text{Nat } \to \text{Nat}_\bot$. $\text{Nat}$ is a pointed CPO, and so is $\text{Nat } \to \text{Nat}_\bot$. The correct functional is:

$$F : (\text{Nat } \to \text{Nat}_\bot) \to (\text{Nat } \to \text{Nat}_\bot)$$

$$F = \lambda f. \lambda n. n \text{ equals zero } \rightarrow \text{ one } \sqcup n \text{ times } \left( f \left( n \text{ minus one} \right) \right)$$

The least fixed point of $\text{fac}$ is

$$\text{fix } F = \bigsqcup \{ \text{fac}_i \mid i \geq 0 \}, \text{ where }$$

$$\text{fac}_0 = (\lambda n. \bot) = \bot$$

$$\text{fac}_{i+1} = F \left( \text{fac}_i \right), i \geq 0$$

The way we use this definition is operationally (just as we required) by using the expansion $\text{fix } F = F \left( \text{fix } F \right)$. For instance:
\[(\text{fix } F)(\text{three}) = (F(\text{fix } F))(\text{three}) = \left((\lambda f.\lambda n. n \text{ equals zero } \rightarrow \text{one } \sqcup \text{n times } \left(f(\text{n minus one})\right))(\text{fix } F)\right)(\text{three}) = \left((\lambda n. n \text{ equals zero } \rightarrow \text{one } \sqcup \text{n times } \left((\text{fix } F)(\text{n minus one})\right)\right)(\text{three}) = \text{three times } \left((\text{fix } F)(\text{three minus one})\right) = \text{three times } \left((\text{fix } F)(\text{two})\right) = \text{three times } \left((F(\text{fix } F))(\text{two})\right) = \text{three times } \left(\text{two times } \left((\text{fix } F)(\text{one})\right)\right) = \text{three times } \left(\text{two times } \left(\text{one times } \left((\text{fix } F)(\text{zero})\right)\right)\right) = \text{three times } \left(\text{two times } \left(\text{one times } \left((F(\text{fix } F))(\text{zero})\right)\right)\right) = \text{three times } \left(\text{two times } \left(\text{one times one}\right)\right) = \text{six}\]

Each time we have to apply \(\text{fix } F\), we first expand it to \(F(\text{fix } F)\) to unfold the recursion one more time, and apply the resultant function to the argument.

**LOOP SEMANTICS**

Instead of the recursive valuation function \(C\[\text{while } B \text{ do } C\] = \(\lambda s. B \sqcup s \rightarrow (C\[\text{while } B \text{ do } C]\)(\text{fix } F)(\text{zero})\) \(\sqcup s\)\), we will write \(C\[\text{while } B \text{ do } C\] = \(\text{fix } \left((\lambda f.\lambda s. B \sqcup s \rightarrow f(\text{fix } F)(\text{zero})\right)\)(\text{zero})\)\) which is the non-recursive definition we were looking for.

Let us unfold an example loop using this semantics:

\(C\[\text{while } A > 0 \text{ do } B := B + 1; A := A - B\]\)

The functional we need is

\(F = \lambda s. B\[A > 0\]s \rightarrow C\[B := B + 1; A := A - B\]s \sqcup s\)

We can then write \(C\[\text{while } A > 0 \text{ do } B := B + 1; A := A - B\] = \text{fix } F\)

If we start with a store \(s_0 = \{[A] \mapsto \text{six}, [B] \mapsto \text{zero}\}\), we can then derive the final store:
\[(\text{fix } F)s_0\]
\[= (F (\text{fix } F))s_0\]
\[= B\mathbb{I}[A > 0]s_0 \rightarrow (\text{fix } F)(C[B := B + 1; A := A - B]s_0) \sqcup s_0\]
\[= (\text{fix } F)(C[B := B + 1; A := A - B]s_0)\]
\[= (\text{fix } F)s_1, \text{ where } s_1 = \{[\llbracket A \rrbracket \mapsto \text{five}], [\llbracket B \rrbracket \mapsto \text{one}]\}\]
\[= (F (\text{fix } F))s_1\]
\[= B\mathbb{I}[A > 0]s_1 \rightarrow (\text{fix } F)(C[B := B + 1; A := A - B]s_1) \sqcup s_1\]
\[= (\text{fix } F)(C[B := B + 1; A := A - B]s_1)\]
\[= (\text{fix } F)s_2, \text{ where } s_2 = \{[\llbracket A \rrbracket \mapsto \text{three}], [\llbracket B \rrbracket \mapsto \text{two}]\}\]
\[= (F (\text{fix } F))s_2\]
\[= B\mathbb{I}[A > 0]s_2 \rightarrow (\text{fix } F)(C[B := B + 1; A := A - B]s_2) \sqcup s_2\]
\[= (\text{fix } F)(C[B := B + 1; A := A - B]s_2)\]
\[= (\text{fix } F)s_3, \text{ where } s_3 = \{[\llbracket A \rrbracket \mapsto \text{zero}], [\llbracket B \rrbracket \mapsto \text{three}]\}\]
\[= (F (\text{fix } F))s_3\]
\[= B\mathbb{I}[A > 0]s_3 \rightarrow (\text{fix } F)(C[B := B + 1; A := A - B]s_3) \sqcup s_3\]
\[= s_3\]

At each stage we use the fixed point property to unfold the recursion, until the test fails, and the final store results.

**THE INFINITE LOOP**

It is important that the semantics of an infinite loop be correct. Consider the loop \textbf{while} true \textbf{do} nop. Its denotation is

\[C[\text{while} \ true \ \text{do} \ nop] = \text{fix}(\lambda f. \lambda s. B[true]s \rightarrow f(C[nop]s) \sqcup s)\]

which reduces to

\[\text{fix}(\lambda f. \lambda s. f s)\]

Since \(\text{fix } F = \bigsqcup \{F^i(\bot) | i \geq 0\}\), we have:

\[F^0 = \lambda s. \bot\]
\[F^1 = F(F^0) = (\lambda f. \lambda s. f s)(\lambda s. \bot) = \lambda s. (\lambda s. \bot) s = \lambda s. \bot\]
\[F^i = \lambda s. \bot\]

So the least upper bound is just the "do nothing" function \(\lambda s. \bot\), as we would expect. In our previous language we used the command \textbf{diverge} to stand in for the infinite loop. It had the same denotation as our proper infinite loop, so our least fixed point semantics fits with our previous thinking.