Our goal is to understand least fix-point semantics of while loops in the language from figure 7.2, page 142 in Schmidt.

Let’s start with a program: \texttt{begin var X; X := 0; while X < 3 do X := X + 1 end}.

We first need to add a valuation function for Boolean expressions to the language. Since we don’t want to add the Boolean domain to the primitives, we’ll just use the C trick of allowing zero to stand for false and anything else for true. In addition, we’ll make the syntax really simple:

Syntax:
\[
B ::= I < N
\]

Valuation:
\[
B : \text{Boolean-expr} \rightarrow \text{Environment} \rightarrow \text{Store} \rightarrow \text{Expressible-value}
\]
\[
B[I < N] = \lambda e.\lambda s. \text{cases} (\text{E}[I]e s) \text{ of}
\]
\[
isNat(n_1) \rightarrow (n_1 \text{ lessthan } N[N] \rightarrow \text{inNat}(\text{one}) \mid \text{inNat}(\text{zero}))
\]
\[
\mid \text{isErrvalue()} \rightarrow \text{inErrvalue()} \text{ end}
\]

Now we can give our program to \( P \) with the first available location, given by \texttt{first-locn}, and an empty store, created by \texttt{newstore}. (Although \( P \) is not defined as a function of \( s \), its functionality indicates this.) Thus:

\[
P[\texttt{begin \ldots end}].\texttt{first-locn newstore}
\]
\[
\Rightarrow K[\texttt{begin \ldots end}](\texttt{emptyenv first-locn newstore}),
\] removing the period, and applying the function of \( l \) to \texttt{first-locn} (\texttt{newstore} is just copied through, for now, waiting for a function to which it can be applied)

Applying \( K \) we get:

\[
\Rightarrow C[X := 0; \texttt{while \ldots }](\texttt{var X})(\texttt{emptyenv first-locn}) \texttt{ newstore}
\]

Now, we do the declaration:

\[
\texttt{D[var X]}(\texttt{empty-env first-locn}) =\]
\[
[X \mapsto (\texttt{next-locn first-locn})(\texttt{emptyenv first-locn})]
\]
using the definitions of \texttt{updateenv} and \texttt{reserve-locn}.

Call this environment \( e_X \), in which \( X \) is bound to a reserved location, called \( l_X \).
We now have an environment for the evaluation of the two commands that form the program. The sequence is evaluated thus:

\[
C[X := 0; \text{while} \ldots]e_X \xrightarrow{\text{newstore}} ((\text{check} (C[\text{while} \ldots]e_X)) \circ (C[X := 0]e_X)) \text{ newstore}
\]

This can be rewritten (since we have the newstore argument) as:

\[
((\text{check} (C[\text{while} \ldots]e_X)) (C[X := 0]e_X \text{ newstore}))
\]

Let’s do the assignment first:

\[
\xrightarrow{s} (\lambda s. \text{cases}(\text{accessenv}[X]e_X)\text{of} \\
\quad \text{isLocation}(l) \rightarrow (\text{cases}(E[0]e_Xs)\text{of} \\
\quad \quad \text{isNat}(n) \rightarrow (\text{return}(\text{update} l n s)) \\
\quad \quad \quad \mid \text{isErrvalue()} \rightarrow (\text{signalerr} s)\text{end})\text{newstore}
\]

\[
\xrightarrow{s} (\text{true}, [l_X \mapsto \text{zero}]\text{newstore})
\]

since \text{accessenv} returns X’s location, E evaluates the constant 0 through N to be \text{zero} and update creates a store in which \(l_X\) is bound to \text{zero}. Notice that the store is tagged as OK by return. This is passed to check which looks at the tag, and applies its argument (the evaluation of the while loop) to the store with the tag taken off.

Since our store is tagged as OK, we need to evaluate the while loop, then apply the function that is returned to the store which has X’s location bound to zero, i.e. to \([l_X \mapsto \text{zero}]\text{newstore}\). Call this store \(s_0\).

Now, the heart of the derivation: the while loop.

\[
C[\text{while} X < 3 \text{ do } X := X + 1]e_X
\]

\[
\xrightarrow{\text{fix}} (\lambda f. \lambda s. \text{cases}(B[X < 3]e_Xs)\text{of} \\
\quad \text{isNat}(t) \rightarrow (t \text{ equals one} \rightarrow (\text{check} f) \circ (C[X := X + 1]e_X) \mid \text{return}(s)) \\
\quad \quad \mid \text{isErrvalue()} \rightarrow (\text{signalerr} s)\text{end})
\]

Note, we have altered the equation for the while loop to accomodate our C ‘trick’ with Boolean expressions evaluating to \text{one} or \text{zero}. Since the body of the function given to fix is a function of s, we cannot evaluate the Boolean test completely since we don’t know which store it needs. However, we can express it as a conditional expression from our semantic equation for B, i.e.:

\[
B[X < 3]e_X = \lambda s. \text{cases} E[X]e_X s \\
\quad \text{isNat}(n) \rightarrow (n \text{ less than } N[3] \rightarrow \text{inNat}(one) \mid \text{inNat}(zero)) \\
\quad \quad \mid \text{isErrvalue()} \rightarrow \text{inErrvalue()}\text{end}
\]

Since X has a location, the evaluation of the identifier reduces to \text{inNat(access l_X s)} so that the test evaluation reduces to \(\lambda s.((\text{access l_X s}) \text{ less than three} \rightarrow \text{inNat}(one) \mid \text{inNat}(zero))\).
The evaluation of the loop can then be written:

\[
\text{fix}(\lambda f. \lambda s. \begin{cases} 
((\text{access } l X s \lesssim \text{three} \rightarrow \text{one } \mid \text{zero}) \equiv \text{one} \rightarrow 
(\text{check } f) \circ (C[lX := X + 1]e_X) \parallel \text{return})(s) 
\end{cases})
\]

The assignment, since \(X\) has a location in \(e_X\) is:

\[
(\lambda s. \begin{cases} 
\text{cases } (E[lX := X + 1]e_X s) \text{ of} \\
\text{isNat}(n) \rightarrow (\text{return } (\text{update } l n s)) \\
\text{isErrorvalue()} \rightarrow (\text{signalerr } s) \text{ end}
\end{cases})
\]

The expression \(E[lX := X + 1]e_X s\) evaluates to \(\text{inNat}((\text{access } l X s \text{ plus one}) \text{ in } \text{ok})\) since \(E[lX := X + 1]e_X s\) gives \(\text{inNat}(\text{access } l X s)\)
and \(E[1]\) gives \(\text{inNat}(\text{one})\).

The assignment is then:

\[
\lambda s. (\text{return } (\text{update } l X ((\text{access } l X s \text{ plus one}) s)), \text{ or} \\
\lambda s. \text{inOK}([lX \mapsto ((\text{access } l X s \text{ plus one}) s] s)
\]

The fix expression is then:

\[
\text{fix}(\lambda f. \lambda s. \begin{cases} 
((\text{access } l X s \lesssim \text{three} \rightarrow \text{one } \mid \text{zero}) \equiv \text{one} \rightarrow 
((\text{check } f) \circ \lambda s. \text{inOK}([lX \mapsto ((\text{access } l X s \text{ plus one}) s]) s) \parallel \text{return})(s)
\end{cases})
\]

Rewrite this as \(f(x)(g)\), where \(g = \lambda f. \lambda s. \ldots\)

Now we apply the fix-point form to undwind the recursion one level:

\[\Rightarrow g(fix(g))\]

If we apply \(g\) to \(fix(g)\), we get:

\[
\lambda s. ((\text{access } l X s \lesssim \text{three} \rightarrow \text{one } \mid \text{zero}) \equiv \text{one} \rightarrow ((\text{check } fix(g)) \circ \lambda s. \text{inOK}(\ldots)) \parallel \text{return})(s)
\]

Recall now that this function is to be applied to the initial store, \(s_0\). If we do this, we get:

\[
\Rightarrow ((\text{check } fix(g)) \circ \lambda s. \text{inOK}(\ldots)) s_0 \\
= ((\text{check } fix(g)) (\lambda s. \text{inOK}(\ldots)) s_0) \\
= ((\text{check } fix(g)) \text{ inOK}(s_1)),
\]

where \(s_1\) is the store in which \(l_X\) is bound to \(\text{one}\), i.e. \([lX \mapsto ((\text{access } l X s_0 \text{ plus one}) s_0), \text{ or } [lX \mapsto \text{one}] s_0\).

Clearly we can unwind the recursion by expanding \(fix(g)\) any number of times, but when we apply the resultant function of \(s\) to the actual store \(s_3\) (in which \(l_X\) is bound to \(\text{three}\)), then we simply get a tagged store, through \(\text{return}\). The loop has thus exited, and we have the least fixed pointed solution for the while loop’s function.

The final result is then \(\text{inOK}(s_3)\), where \(s_3\) is the expression:
\[ l_X \mapsto three[l_X \mapsto two][l_X \mapsto one][l_X \mapsto zero]newstore, \text{ where } l_X = (next-locn \ first-locn). \]

This clearly reduces (by the extensionality principle) to :

\[ (next-locn \ first-locn) \mapsto three]newstore \]

Notice that we were only able to compute the final store by giving \( P \) the empty store \( newstore \). The meaning of the program, as a function of \( s \) is a horrendous expression full of suspended evaluations waiting for an actual store. Thus derivations like this one are much easy to write out than static meanings.