datatype 'a buffer = Buf of {
    data : 'a option ref,
    emptySem : semaphore,
    fullSem : semaphore
}

fun buffer () = BUF{
    data = ref NONE,
    emptySem = semaphore 1,
    fullSem = semaphore 0
}
fun insert (BUF{data, emptySem, fullSem}, v) = (  
  P emptySem;  
  data := SOME v;  
  V fullSem)

fun remove (BUF{data, emptySem, fullSem}) = let  
  val _ = P fullSem  
  val (SOME v) = !data
in  
  data := NONE;  
  V emptySem;  
  v
end

Invariant: (emptySem = 1) => (((!data = NONE) and (fullSem = 0))  
Invariant: (fullSem = 1) => (!data <> NONE)
Producer/Consumer buffer using Mutex Locks (1)

```ocaml
datatype 'a buffer = BUF of {
    data : 'a option ref,
    mu : mutex,
    dataAvail : condition,
    dataEmpty : condition
}

fun buffer () = let
    val mu = mutex()
    in
        BUF{
            data = ref NONE,
            mu = mu,
            dataAvail = condition mu,
            dataEmpty = condition mu
        }
    end
```
Producer/Consumer buffer using Mutex Locks (2)

fun insert (BUF{data, mu, dataAvail, dataEmpty}, v) = let
  fun waitLp () = (case !data
     of NONE => (data := SOME v; signal dataAvail)
      | (SOME v) => (wait dataEmpty; waitLp(!data))
      (* end case *))
  in
    withLock mu waitLp ()
  end

fun remove (BUF{data, mu, dataAvail, dataEmpty}) = let
  fun waitLp () = (case !data
     of NONE => (wait dataAvail; waitLp(!data))
      | (SOME v) => (data := NONE; signal dataEmpty; v)
      (* end case *))
  in
    withLock mu waitLp ()
  end
Producer/Consumer buffer using Asynchronous Message Passing (1)

```haskell
datatype 'a buffer = BUF of {
  emptyCh : unit channel,
  insCh   : 'a channel,
  remCh   : 'a channel,
  remAckCh : unit channel,
} 

fun insert (BUF{insCh, emptyCh, ...}, v) = (
  recv emptyCh; send(insCh, v))

fun remove (BUF{remCh, remAckCh, ...}) = (
  (recv remCh) before send(remAckCh, ()))
```
fun buffer () = let
val emptyCh = channel() and insCh = channel()
val remCh = channel() and remAckCh = channel()
fun empty () = (
    send (emptyCh, ());
    full (recv insCh))
and full x = (
    send (remCh, x);
    empty (recv remAckCh))
in
    spawn empty;
    BUF{
        emptyCh = emptyCh, insCh = insCh,
        remCh = remCh, remAckCh = remAckCh
    }
end
Producer/Consumer buffer using Synchronous Message Passing

```plaintext
datatype 'a buffer = BUF of {
insCh    : 'a channel,
remCh    : 'a channel
}

fun buffer () = let
    val insCh = channel() and remCh = channel()
    fun empty () = full (recv insCh)
    and full x = (send (remCh, x); empty())
    in
        spawn empty;
        BUF{insCh = insCh, remCh = remCh}
    end

fun insert (BUF{insCh, ...}, v) = send(insCh, v)

fun remove (BUF{remCh, ...}) = recv remCh
```

Unbounded Producer/Consumer buffer using Synchronous Message Passing

datatype 'a buffer = BUF of {
    insCh : 'a channel,
    remCh : 'a channel
}

fun buffer () = let
    val insCh = channel() and remCh = channel()
    fun loop [] = loop [recv insCh]
    | loop (buf as x::r) = select[
        wrap(sendEvt(remCh, x), fn () => loop r),
        wrap(receiveEvt insCh, fn y => loop(y::buf))
    ]
    in
        spawn empty;
        BUF{insCh = insCh, remCh = remCh}
    end

fun insert (BUF{insCh, ...}, v) = send(insCh, v)

fun remove (BUF{remCh, ...}) = recv remCh