1 Higher Order Example

1.1 Program 1

```
resource File

def writeFile(v: Int): Unit with File.write =
  File.write(v)

def doIt(f : Int → Unit with ∅): Unit =
  f(0)

def unlabeled(): Unit =
  doIt(writeFile)

unlabeled()
```

Globally this gives the correct result (the effect is `File.write` but locally is not correct because `writeFile` is being passed to `doIt`, and there is a mismatch between the effect signatures.

1.2 Program 2

If we de-sugar the program by “one layer”, we get the following.

```

let x_1 = new_σ x ⇒ {
  def writeFile(v: Int): Unit with File.write =
    File.write(v)
} in

let x_2 = new_σ x ⇒ {
  def doIt(obj: {f: Int → Unit with ∅}): Unit with ∅ =
    f(0)
} in

let x_3 = new_d x ⇒ {
  def unlabelled(): Unit =
    x_2.doIt(x_1)
} in

x_3.unlabelled()
```

To typecheck x_3 your choice of Γ' will need both x_1 and x_2. Now, $\texttt{capture}(x_1) = \texttt{File.write}$ and $\texttt{capture}(x_2) = \varnothing$. Therefore $\texttt{capture}(\Gamma') = \varepsilon_c = \texttt{File.write}$. However, in the premise of C-NewObj, $\texttt{capture}(x_2) \supseteq \varepsilon_c$ is NOT true so it wouldn't typecheck. Since your choice of Γ' needs at least x_2 to be well-formed, then it won't typecheck under any choice of Γ'.

1.3 Program 3

If we translate the let expressions we get the following:

```
new_σ x ⇒ {
  def _dummy1(x_1: { writeFile: Int → Unit with File.write }): Unit with File.write =

    new_d x ⇒ {
      def _dummy2(x_2: {doIt: {f: Int → Unit with ∅} → Unit}): Unit =

        new_d x ⇒ {
          def _dummy3(x_3: {unlabelled: Unit → Unit}): Unit =
            x_3.unlabelled()

```

```
              }._dummy3(new$_d$x ⇒ { def unlabelled(): Unit =
                                      $x_2$.doIt($x_1$) })

       }._dummy2(new$_d$x ⇒ { def doIt(obj: { f: Int → Unit with ∅ }): Unit =
                              f(0) })

}._dummy1(new$_\sigma$x ⇒ { def writeFile(v: Int): Unit with File.write =
                      File.write(v) })
```

2 Weirdness Without Well-Formedness

```
let $x_1$ = new$_\sigma$ x ⇒ {
   def example($y_1$: {meth: Unit → Unit with File.write}): Unit with File.write =
      $y_1$.meth()
} in

let $x_2$ = new$_\sigma$ x ⇒ {
   def meth($y_2$: Unit): Unit with ∅ = unit
} in

$x_1$.example($x_2$)
```

This program e is a counter-example to the (naive) Use Lemma because $\varnothing \vdash e : \texttt{Unit}\ \textit{with}\ \texttt{File.write}$, but $\texttt{File.write} \not\subseteq \texttt{capture}(\varnothing)$. Two possible solutions:

1. The type system does a check to make sure types only reference known resources (which would require you to typecheck in the context $\Gamma = \texttt{File} : \{\texttt{File}\}$).
2. Add a condition to the Use Lemma like "the program under consideration is closed under Γ", where a suitable definition of closed would mean that the program above, typechecked in $\varnothing$, doesn't count because File occurs free in the type of y_1.

3 Higher-Order W/ Currying

3.1

```
def go(a: Int, b: Int): Unit with File.write =
    File.write

def fixsecond(f: Int → Unit with ∅): Unit with ∅ =
    f(0)

def fixfirst(f: Int → Int → Unit with ∅): (Int → Unit with ∅) with ∅ =
    fixsecond(f(0))

def main(): Unit =
    fixfirst(go)
```

3.2

```
let $x_1$ = new$_\sigma$x ⇒ {
    def go(a: Int): {f: Int → Unit with File.write} with ∅ =
       new$_\sigma$x ⇒ {
           def go(b: Int): Unit with File.write =
              File.write
       }
} in

let $x_2$ = new$_\sigma$x ⇒ {
    def fixsecond(obj: {go: Int → Unit with ∅}): Unit with ∅ =
        obj.go(0)
}

let $x_3$ = new$_\sigma$x ⇒ {
    def fixfirst(obj: {go: Int → Int → Unit with ∅}): {go: Int → Unit} with ∅ =
        obj.go(0)
} in

let $x_4$ = new$_d$x ⇒ {
    def main(): Unit =
        $x_3$.fixfirst($x_1$)
} in

$x_4$.main()
```