1 Basic Effect Polymorphism

Pseudo-Wyvern

```
def polymorphicWriter(x: T <: {File, Socket}): Unit with T.write =
    x.write

/* below invocation should typecheck with File.write as its only effect */
polymorphicWriter File
```

λ-Calculus

```
let pw = λϕ ⊆ {File.write, Socket.write}.
   λf: Unit →ϕ Unit.
      f unit

in let makeWriter = λr: {File, Socket}.
   λx: Unit. r.write

in (pw {File.write}) (makeWriter File)
```

Typing

To type the definition of `polymorphicWriter`:

1. By ε-App
   $\phi \subseteq \{\texttt{F.w}, \texttt{S.w}\}$, x: $\texttt{Unit} \to_\phi \texttt{Unit} \vdash x\ \texttt{unit} : \texttt{Unit with}\ \phi$.
2. By ε-Abs
   $\phi \subseteq \{\texttt{F.w}, \texttt{S.w}\} \vdash \lambda x : \texttt{Unit} \to_\phi \texttt{Unit}.x\ \texttt{unit} : (\texttt{Unit} \to_\phi \texttt{Unit}) \to_\phi \texttt{Unit with}\ \varnothing$
3. By ε-PolyFxAbs,
   $\vdash \forall\phi \subseteq \{\texttt{S.w}, \texttt{F.w}\}.\lambda x : \texttt{Unit} \to_\phi \texttt{Unit}.x\ \texttt{unit} : \forall\phi \subseteq \{\texttt{F.w}, \texttt{S.w}\}.(\texttt{Unit} \to_\phi \texttt{Unit}) \to_\phi \texttt{Unit caps}\ \varnothing\ \texttt{with}\ \varnothing$

Then (`pw {File.write}`) can be typed as such:

4. By ε-PolyFxApp,
   $\vdash \texttt{pw}\ \{\texttt{F.w}\} : [\{\texttt{F.w}\}/\phi]((\texttt{Unit} \to_\phi \texttt{Unit}) \to_\phi \texttt{Unit})\ \texttt{with}\ [\{\texttt{F.w}\}/\phi]\varnothing \cup \varnothing$

The judgement can be simplified to:

5. $\vdash \texttt{pw}\ \{\texttt{F.w}\} : (\texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}) \to_{\{\texttt{F.w}\}} \texttt{Unit with}\ \varnothing$

Any application of this function, as in (`pw {File.write}`)(`makeWriter File`), will therefore type as having the single effect `F.w` by applying ε-App to judgement (5).

2 Dependency Injection

Pseudo-Wyvern

An HTTPServer module provides a single `init` method which returns a `Server` that responds to HTTP requests on the supplied socket.

```
module HTTPServer

def init(out: A <: {File, Socket}): Str →A.write Unit with ∅ =
   λ msg: Str.
      if (msg == ‘‘POST’’) then out.write(‘‘post response’’)
      else if (msg == ‘‘GET’’) then out.write(‘‘get response’’)
      else out.write(‘‘client error 400’’)
```

The main module calls `HTTPServer.init` with the `Socket` it should be writing to.

```
module Main
require HTTPServer, Socket

def main(): Unit =
   HTTPServer.init(Socket) ‘‘GET /index.html’’
```

The testing module calls `HTTPServer.init` with a `LogFile`, perhaps so the responses of the server can be tested offline.

```
module Testing
require HTTPServer, LogFile

def testSocket(): =
   HTTPServer.init(LogFile) ‘‘GET /index.html’’
```

λ-Calculus

The HTTPServer module:

```
MakeHTTPServer = λx: Unit.
   λϕ ⊆ {LogFile.write, Socket.write}.
      λf: Str →ϕ Unit.
         λmsg: Str.
            f msg
```

The Main module:

```
MakeMain = λhs: HTTPServer. λsock: {Socket}.
   λx: Unit.
      let socketWriter = (λs: {Socket}. λx: Unit. s.write) sock in
      let theServer = hs {Socket.write} socketWriter in
      theServer ‘‘GET/index.html’’
```

The Testing module:

```
MakeTest = λhs: HTTPserver. λlf: {LogFile}.
   λx: Unit.
      let logFileWriter = (λl: {LogFile}. λx: Unit. l.write) lf in
      let theServer = hs {LogFile.write} logFileWriter in
      theServer ‘‘GET/index.html’’
```

A single, desugared program for production would be:

```
let MakeHTTPServer = λx: Unit.
   λϕ ⊆ {LogFile.write, Socket.write}.
      λf: Str →ϕ Unit.
         λmsg: Str.
            f msg

in let Run = λSocket: {Socket}.
   let HTTPServer = MakeHTTPServer unit in
   let Main = MakeMain HTTPServer Socket in
   Main unit

in Run Socket
```

A single, desugared program for testing would be:

```
let MakeHTTPServer = λx: Unit.
   λϕ ⊆ {LogFile.write, Socket.write}.
      λf: Str →ϕ Unit.
         λmsg: Str.
            f msg

```

```
in let Run = λLogFile: {LogFile}.
   let HTTPServer = MakeHTTPServer unit in
   let Main = MakeMain HTTPServer LogFile in
   Main unit

in Run LogFile
```

Note how the HTTPServer code is identical in the testing and production examples.

Typing

```
let MakeHTTPServer = λx: Unit.
   λϕ ⊆ {LogFile.write, Socket.write}.
      λf: Str →ϕ Unit.
         λmsg: Str.
            f msg
```

To type `MakeHTTPServer`:

1. By ε-App,
   $\texttt{x} : \texttt{Unit},\ \phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}, \texttt{f} : \texttt{Str} \to_\phi \texttt{Unit},\ \texttt{msg} : \texttt{Str}$
   $\vdash \texttt{f msg} : \texttt{Unit with } \phi$
2. By ε-Abs,
   $\texttt{x} : \texttt{Unit},\ \phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}, \texttt{f} : \texttt{Str} \to_\phi \texttt{Unit}$
   $\vdash \lambda\texttt{msg} : \texttt{Str}.\ \texttt{f msg} : \texttt{Str} \to_\phi \texttt{Unit with } \varnothing$
3. By ε-Abs,
   $\texttt{x} : \texttt{Unit},\ \phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}$
   $\vdash \lambda\texttt{f} : \texttt{Str} \to_\phi \texttt{Unit}.\ \lambda\texttt{msg} : \texttt{Str}.\ \texttt{f msg} :$
   $(\texttt{Str} \to_\phi \texttt{Unit}) \to_\varnothing (\texttt{Str} \to_\phi \texttt{Unit}) \texttt{ with } \varnothing$
4. By ε-PolyFxAbs,
   $\texttt{x} : \texttt{Unit}$
   $\vdash \lambda\phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}.\ \lambda\texttt{f} : \texttt{Str} \to_\phi \texttt{Unit}.\ \lambda\texttt{msg} : \texttt{Str}.\ \texttt{f msg} :$
   $\forall\phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}.(\texttt{Str} \to_\phi \texttt{Unit}) \to_\varnothing (\texttt{Str} \to_\phi \texttt{Unit}) \texttt{ caps } \varnothing \texttt{ with } \varnothing$
5. By ε-Abs,
   $\vdash \lambda\texttt{x} : \texttt{Unit}.\ \lambda\phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}.\ \lambda\texttt{f} : \texttt{Str} \to_\phi \texttt{Unit}.\ \lambda\texttt{msg} : \texttt{Str}.\ \texttt{f msg} :$
   $\texttt{Unit} \to_\varnothing \forall\phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}.(\texttt{Str} \to_\phi \texttt{Unit}) \to_\varnothing (\texttt{Str} \to_\phi \texttt{Unit}) \texttt{ caps } \varnothing \texttt{ with } \varnothing$

Note that after two applications of `MakeHTTPServer`, as in `MakeHTTPServer unit {Socket.write}`, it would type as follows:

6. By ε-PolyFxApp,
   $\texttt{x} : \texttt{Unit}$
   $\vdash \texttt{MakeHTTPServer unit } \{\texttt{S.w}\} :$
   $(\texttt{Str} \to_{\{\texttt{S.w}\}} \texttt{Unit}) \to_\varnothing (\texttt{Str} \to_{\{\texttt{S.w}\}} \texttt{Unit}) \texttt{ with } \varnothing$

After fixing the polymorphic set of effects, possessing this function only gives you access to the `Socket.write` effect.

3 Map Function

Pseudo-Wyvern

```
def map(f: A →ϕ B, l: List[A]): List[B] with ϕ =
   if isnil l then []
   else cons (f (head l)) (map (tail l f))
```

λ-Calculus

```
map = λφ. λA. λB.
  λf: A→φB.
    (fix (λmap: List[A] → List[B]).
      λl: List[A].
        if isnil l then []
        else cons (f (head l)) (map (tail l f)))
```

Typing

- This has the type: $\forall\phi.\forall A.\forall B.(A \to_\phi B) \to_\varnothing$ `List`$[A] \to_\phi$ `List`$[B]$ `with` $\varnothing$.
- `map` $\varnothing$ is a pure version of map.
- `map` {`File`.$*$} is a version of map which can perform operations on `File`.

4 Thread Non-Interference

Given two threads, we want to know if they can be executed at the same time without interference. The below example demonstrates two threads which are executing non-interfering code. `fx1` can only have file effects, while `fx2` can only have socket effects. Even before fixing the set of effects, we can tell executing them will not interfere with the other.

```
fx1 = λφ ⊆ {File.∗}. λf : Str →φ Unit
fx2 = λφ ⊆ {Socket.∗}. λf : Str →φ Unit

spawn (fx1 {File.write} e1 ‘‘data2process’’)
spawn (fx2 {Socket.write} e2 ‘‘data2process’’)
```

By contrast, the below example demonstrates how two threads might interfere. `Fx` is polymorphic in its set of effects, which is bound from above by {`File`.$*$}. The first two threads have their effect-variable bound to different, disjoint sets of effects; but the second two threads contain a common effect, `File.write`, and so their execution might interfere.

```
fx = λφ ⊆ {File.∗}. λf : Str →φ Unit

// Non−interfering threads.
spawn (fx {File.write} e1 ‘‘data2process’’)
spawn (fx {File.append} e2 ‘‘data2process’’)

// Potentially interfering threads.
spawn (fx {File.write} e1 ‘‘data2process’’)
spawn (fx {File.write, File.read} e2 ‘‘data2process’’)
```