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

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

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

The judgement can be simplified to:

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

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

3 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

To type `MakeHTTPServer`:

1. By ε-App,
   $\texttt{x} : \texttt{Unit},\ \texttt{Writer} <: \texttt{Str} \rightarrow_{\{\texttt{Lf.write},\texttt{S.write}\}} \texttt{Unit},\ \texttt{f} : \texttt{Writer},\ \texttt{msg} : \texttt{Str}$
   $\vdash \texttt{f msg} : \texttt{Unit}$

Types

- `HTTPServer.init` has the type $\lambda A <: \{\texttt{File}, \texttt{Socket}\}.\ A \rightarrow_{\varnothing} \texttt{Str} \rightarrow_{A.write} \texttt{Unit}$