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 ($\texttt{pw}\ \{\texttt{File.write}\}$) 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 ($\texttt{pw}\ \{\texttt{File.write}\}$)(`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 $\texttt{MakeHTTPServer}$:

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

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

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

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

4 Imports Are an Upper Bound on Polymorphic Capabilities

If you import a polymorphic function with no upper-bound on its effects, then the collective effects of the other capabilities being imported will be an upper-bound. The following should typecheck.

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

import({File.*})
  pw = polywriter
  f = File
in
  pw {File.write} (λx: Unit. f.write)
```

Derivation

$\texttt{ho-safe}(\forall\phi \subseteq \{\texttt{File.write}, \texttt{Socket.write}\}.\texttt{Unit} \to_\phi \texttt{Unit}, \texttt{File}.*)$
$= \{\texttt{File.write}, \texttt{Socket.write}\} \subseteq \{\texttt{File}.*\} \wedge \texttt{safe}(\texttt{Unit} \to_{\{\texttt{File.write},\texttt{Socket.write}\}} \texttt{Unit}, \texttt{File}.*)$

Both the $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s,)$ and $\texttt{effects}(\hat{\tau}_i) \subseteq \varepsilon_s$ check fails, because $\{\texttt{File.write}, \texttt{Socket.write}\} \subseteq \{\texttt{File.write}\}$ is false. If we were to intersect $\texttt{effects}(\hat{\tau}_i)$ with the effects of the capabilities being passed in, we would get a tighter bound on the actual effects of $\hat{\tau}_i$, and the check would succeed. Similar thing with $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s,)$. Recall that $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s,)$ is a version of $\texttt{ho-effects}(\hat{\tau}_i) \subseteq \varepsilon_s$ which distributes over the subterms in $\hat{\tau}_i$. If we intersect the $\texttt{ho-effects}(\hat{\tau}_i)$ with the capabilites passed in, this check would also pass.

How do we define this intersection? Several ways come to mind.

Extra Argument Define a two-variable version of effects which has an extra argument containing all of the imported capabilities. This function will return the effects, but will produce a (potentially) tighter upper-bound using the information from the imported capabilities. This function will be defined the same on non-polymorphic types as the one-variable version, but for polymorphic functions it is defined in the following way:

- $\texttt{ho-effects}(\hat{\tau}_i, \overline{\hat{\tau}}) = \texttt{ho-effects}(\hat{\tau}_i) \cap \bigcup_i \texttt{effects}(\hat{\tau}_i, \overline{\hat{\tau}})$
- $\texttt{effects}(\hat{\tau}_i, \overline{\hat{\tau}}) = \texttt{effects}(\hat{\tau}_i) \cap \bigcup_i \texttt{ho-effects}(\hat{\tau}_i, \overline{\hat{\tau}})$

Do a similar thing for safe and ho-safe. The premises of ε-Import now use the versions with the extra argument.

Extra Premises Modify the premises of ε-Import. Add a new predicate, $\texttt{is-poly}(\hat{\tau})$ which is true iff $\hat{\tau}$ is a polymorphic type. The premises are now:

- $\neg\texttt{is-poly}(\hat{\tau}) \implies \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s$
- $\texttt{is-poly}(\hat{\tau}) \implies \texttt{effects}(\hat{\tau}) \cap \bigcup_i \texttt{effects}(\hat{\tau}_i) \subseteq \varepsilon_s$

How to do the higher-order safety checks though?

Replacement Define $\varepsilon' = \bigcup_i$ $\texttt{effects}(\hat{\tau}_i)$, the sum of the effects in scope. Then define $\texttt{replace}(\lambda\phi \subseteq \varepsilon.\hat{\tau}) = \lambda\phi \subseteq \varepsilon'.\hat{\tau}$. Then for any polymorphic capability $\hat{\tau}_i$¡ instead of doing e.g. $\texttt{effects}(\hat{\tau})$, do $\texttt{effects}(\texttt{replace}(\hat{\tau}, \varepsilon'))$. (How would this work for polymorphic types?)

5 Violating a polymorphic function that has been fixed

Malicious code tries to import polywriter, where the effect-set has been fixed to $\{\texttt{File.write}\}$, and then calls it with $\{\texttt{Socket.write}\}$. The example should reject.

```

let polywriter = λφ ⊆ {File.write, Socket.write}. λf: Unit →_φ Unit. f unit

import({File.*, Socket.*})
  filewriter = polywriter {File.write}
  s = λx: Unit. Socket.write
in
  filewriter s
```

Safely rejects because the higher-order safety check is not true (acknowledging that `filewriter` could be passed a capability exceeding its authority).

$$\texttt{ho-safe}((\texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}) \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}, \{\texttt{File.}*, \texttt{Socket.}*\})$$

$$= \texttt{safe}(\texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}, \{\texttt{File.}*, \texttt{Socket.}*\}) \wedge \texttt{ho-safe}(\texttt{Unit}, \{\texttt{File.}*, \texttt{Socket.}*\})$$

$$= \texttt{safe}(\texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}, \{\texttt{File.}*, \texttt{Socket.}*\})$$

$$= \{\texttt{File.}*, \texttt{Socket.}*\} \subseteq \{\texttt{File.}*\}$$

which is false.

6 Composing polymorphic functions (artificial example)

```
λφ_1 ⊆ { File.write, File.read }.
  λφ_2 ⊆ φ_1.
    λf: Unit →_{φ_1} Unit.
      λg: Unit →_{φ_2} Unit.
        let _ = f unit in g unit
```