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

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

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

The judgement can be simplified to:

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

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 $\texttt{MakeHTTPServer unit } \{\texttt{Socket.write}\}$, it would type as follows:

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

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 Imports Are an Upper Bound on Polymorphic Capabilities

4.1 Example 1

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

import({File.*})
   pw = polywriter
   f = File
in
   e
```

In the unannotated code e, you can never make `pw` return a socket-writing function, because there is no socket-writing capability in scope that it could be given. However, this example should fail for a different reason: there is a file capability in scope, and you could pass `pw` a function which captures any effect on that file, which would violate its signature. For instance:

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

This example should typecheck, since typechecking of the unannotated body strips all annotations from the imported capabilities. However, as of 17/05/2017, there is no way to apply effect-polymorphic types in an unannotated context.

Derivation

For this section we are going to be conflating the name of a variable with its type (so pw really means the type of the variable pw, which is the effect-polymorphic type). Firstly, note that $\texttt{effects}(pw) = \texttt{ho-effects}(pw) = \{\texttt{File.write}, \texttt{Socket.write}\}$. Then:

$$\begin{aligned}
&\texttt{effects}(pw, \{\{\texttt{File}\}\}) \\
&= \texttt{effects}(pw) \cap \texttt{effects}(\{\texttt{File}\}) \\
&= \{\texttt{File.write}, \texttt{Socket.write}\} \cap \{\texttt{File.}*\} \\
&= \{\texttt{File.write}\} \subseteq \varepsilon_s = \{\texttt{File.}*\}
\end{aligned}$$

And also:

$$\begin{aligned}
&\texttt{effects}(\{\texttt{File}\}, \{pw\}) \\
&= \texttt{effects}(\{\texttt{File}\}) \\
&= \{\texttt{File.}*\} \subseteq \varepsilon_s = \{\texttt{File.}*\}
\end{aligned}$$

However, $\texttt{ho-safe}(pw, \varepsilon_s,)$ will fail, causing this example to not typecheck.

$\texttt{ho-safe}(pw, \varepsilon_s)$
$= \texttt{ho-safe}(\forall\phi \subseteq \{\texttt{File.write}, \texttt{Socket.write}\}.((\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\phi \texttt{Unit})\ \texttt{caps}\ \varnothing, \{\texttt{File.*}\})$
$= \varnothing \subseteq \{\texttt{File.*}\} \wedge \texttt{safe}(((\texttt{Unit} \rightarrow_{\{F.w,S.w\}} \texttt{Unit}) \rightarrow_{\{F.w,S.w\}} \texttt{Unit}), \{\texttt{File.*}\})$
$= \{\texttt{File.*}\} \subseteq \{\texttt{File.write}, \texttt{Socket.write}\} \wedge \ldots\ldots$

The last line is not true, because $\{\texttt{File.*}\} \subseteq \{\texttt{File.write}, \texttt{Socket.write}\}$ is not true. The inutition here is that it is failing because you might pass some capability into pw which does any file operation — and pw only permits it to be writing.

4.2 Example 2

This is a modified version of the above example. Instead of passing in a `File`, we pass in a restricted capability that only endows its bearer with write operations on a `File`. This modified version should safely typecheck. The point is that, although the polymorphic function could theoretically be applied so that it returns a socket-writing function, this can't be done in practice because no socket-writing capability can be given to it. It's therefore safe to leave `Socket.write` out of the selected authority.

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

let fwriter = λx: Unit. File.write

import({File.write})
  pw = polywriter
  fw = fwriter
in
  pw {File.write} fw
```

Now we can verify that it meets the conditions of ε-Import. Firstly, note that $\texttt{effects}(pw) = \texttt{ho-effects}(pw) = \{\texttt{File.write}, \texttt{Socket.write}\}$, and $\texttt{effects}(fw) = \{\texttt{File.write}\}$ and $\texttt{ho-effects}(fw) = \varnothing$.

$\texttt{effects}(pw, \{fw\})$
$= \texttt{effects}(pw) \cap \texttt{effects}(fw)$
$= \{\texttt{File.write}, \texttt{Socket.write}\} \cap \{\texttt{File.write}\}$
$= \{\texttt{File.write}\} \subset \varepsilon_s = \{\texttt{File.write}\}$

And also

$\texttt{effects}(fw, \{pw\})$
$= \texttt{effects}(fw)$
$= \{\texttt{File.write}\} \subseteq \varepsilon_s = \{\texttt{File.write}\}$

Next we shall check that $\texttt{ho-safe}(pw, \varepsilon_s)$ and $\texttt{ho-safe}(fw, \varepsilon_s)$.

$\texttt{ho-safe}(pw, \varepsilon_s)$
$= \texttt{ho-safe}(\forall\phi \subseteq \{\texttt{File.write}, \texttt{Socket.write}\}.((\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\phi \texttt{Unit})\ \texttt{caps}\ \varnothing, \{\texttt{File.write}\})$
$= \varnothing \subseteq \{\texttt{File.write}\} \wedge \texttt{safe}(((\texttt{Unit} \rightarrow_{\{\texttt{F.w,S.w}\}} \texttt{Unit}) \rightarrow_{\{\texttt{F.w,S.w}\}} \texttt{Unit}), \{\texttt{File.write}\})$
$= \texttt{safe}(((\texttt{Unit} \rightarrow_{\{\texttt{F.w,S.w}\}} \texttt{Unit}) \rightarrow_{\{\texttt{F.w,S.w}\}} \texttt{Unit}), \{\texttt{File.write}\})$
$= \{\texttt{File.write}\} \subseteq \{\texttt{File.write}, \texttt{Socket.write}\} \wedge \texttt{ho-safe}(\texttt{Unit} \rightarrow_{\{\texttt{F.w,S.w}\}} \texttt{Unit}, \{\texttt{File.write}\}) \wedge \texttt{safe}(\texttt{Unit}, \{\texttt{File.write}\})$
$= \texttt{ho-safe}(\texttt{Unit} \rightarrow_{\{\texttt{F.w,S.w}\}} \texttt{Unit}, \{\texttt{File.write}\})$
$= \texttt{safe}(\texttt{Unit}, \{\texttt{F.w}, \texttt{S.w}\})$
$= \texttt{true}$

$\texttt{ho-safe}(fw, \varepsilon_s)$
$= \texttt{ho-safe}(\texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}, \{\texttt{File.write}\})$
$= \texttt{safe}(\texttt{Unit}, \{\texttt{File.write}\}) \wedge \texttt{ho-safe}(\texttt{Unit}, \{\texttt{File.write}\})$
$= \texttt{true}$
So it successfully accepts.

5 Violating a polymorphic function that has been fixed

Malicious code tries to import polywriter, where the effect-set has been fixed to {`File.write`}, and then calls it with {`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
```