# 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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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 $\varepsilon$-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: $\forall\phi.\forall A.\forall B.(A \to_\phi B) \to_\varnothing \texttt{List}[A] \to_\phi \texttt{List}[B]$ `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.

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

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.

```
1  let polywriter = λφ ⊆ {File.write, Socket.write}. λf: Unit →φ Unit. f unit
2
3  let fwriter = λx: Unit. File.write
4
5  import({File.write})
6     pw = polywriter
7     fw = fwriter
8  in
9     pw {File.write} fw
```

Now we can verify that it meets the conditions of $\varepsilon$-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 $\texttt{ho-safe}(fw, \varepsilon_s)$.

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

$\texttt{ho-safe}(fw, \varepsilon_s)$
$= \texttt{ho-safe}(\texttt{Unit} \to_{\{\texttt{File.write}\}} \texttt{Unit}, \{\texttt{File.write}\})$
$= \texttt{safe}(\texttt{Unit}, \{\texttt{File.write}\}) \wedge \texttt{ho-safe}(\texttt{Unit}, \{\texttt{File.write}\})$
$= \texttt{true}$
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.

```
1
2  let polywriter = λφ ⊆ {File.write, Socket.write}. λf: Unit →φ Unit. f unit
3
4  import({File.*, Socket.*})
5     filewriter = polywriter {File.write}
6     s = λx: Unit. Socket.write
7  in
8     filewriter s
```

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

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

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

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

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

which is false.

## 6 Polywriter Can Only File.write

Imported capabilities:

- $\texttt{fw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{File.write}$
- $\texttt{pw} = \lambda \Phi \subseteq \{\texttt{File.}*\}.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{funit})$
- $\varepsilon_s = \{\texttt{File.write}\}$

Types:

- $\texttt{type}(\texttt{fw}) = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{pw}) = \forall \Phi \subseteq \{\texttt{F.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \varnothing$

Effects

- $\texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{F.w}\}$
- $\texttt{effects}(\texttt{type}(\texttt{pw}))$
  $= \varnothing \cup \texttt{effects}((\texttt{Unit} \to_{\varnothing} \texttt{Unit}) \to_{\varnothing} \texttt{Unit})$
  $= \varnothing$
- $\texttt{effects}(\hat{\tau}_{\texttt{i}}) \subseteq \varepsilon_{\texttt{s}} = \texttt{True}$

Higher-Order Safety

- $\texttt{ho-safe}(\texttt{type}(\texttt{fw}), \varepsilon_s)$
  $\texttt{ho-safe}(\texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}, \{\texttt{F.w}\})$
  $= \texttt{True}$
- $\texttt{ho-safe}(\texttt{type}(\texttt{pw}), \varepsilon_s)$
  $= \texttt{ho-safe}(\forall \Phi \subseteq \{\texttt{F.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \varnothing, \{\texttt{F.w}\})$
  $= \{\texttt{F.w}\} \subseteq \{\texttt{F.}*\} \wedge \texttt{ho-safe}((\texttt{Unit} \to_{\{\texttt{F.}*\}} \texttt{Unit}) \to_{\{\texttt{F.}*\}} \texttt{Unit}, \{\texttt{F.w}\})$
  $= \texttt{safe}(\texttt{Unit} \to_{\{\texttt{F.}*\}} \texttt{Unit}, \{\texttt{F.w}\}) \wedge \texttt{ho-safe}(\texttt{Unit}, \{\texttt{F.w}\})$
  $= \{\texttt{F.w}\} \subseteq \{\texttt{F.}*\}$
  $= \texttt{True}$

## 7 Polywriter Captures an Effect

Imported capabilities:

- $\texttt{fw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{File.write}$
- $\texttt{pw} = \lambda \Phi \subseteq \{\texttt{File}.*\}.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{Socket.write};\ \texttt{f unit})$
- $\varepsilon_s = \{\texttt{File.write}, \texttt{Socket.write}\}$

Types:

- $\texttt{type}(\texttt{fw}) = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{pw}) = \forall \Phi \subseteq \{\texttt{F}.*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi \cup \{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing$

Effects

- $\texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{F.w}\}$
- $\texttt{effects}(\texttt{type}(\texttt{pw}))$
  $= \varnothing \cup \texttt{effects}((\texttt{Unit} \to_{\varnothing} \texttt{Unit}) \to_{\{\texttt{S.w}\}} \texttt{Unit})$
  $= \{\texttt{S.w}\}$
- $\texttt{effects}(\hat{\tau}_{\texttt{i}}) \subseteq \varepsilon_{\texttt{s}} = \texttt{True}$

Higher-Order Safety

- $\texttt{ho-safe}(\texttt{type}(\texttt{fw}), \varepsilon_s)$
  $\texttt{ho-safe}(\texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \texttt{True}$
- $\texttt{ho-safe}(\texttt{type}(\texttt{pw}), \varepsilon_s)$
  $= \texttt{ho-safe}(\forall \Phi \subseteq \{\texttt{F}.*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi \cup \{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \{\texttt{F.w}, \texttt{S.w}\} \not\subseteq \{\texttt{F}.*\}$
  $= \texttt{False}$

So this example is not higher-order safe. The problem is when you have an example like this:

```
1
2  /* Make a unit → unit that incurs F.w and S.w */
3  let tricky = λx:Unit. (pw {F.w} fw)
4
5  /* Annotated version of pw only allowed to receive a function that does at most F.*.
6     But tricky does {F.w, S.w}, and this will typecheck as unannotated code. */
7  in (pw {F.w} tricky)
```

To get it to typecheck, you need to update the upper-bound on `pw` to be:

- $\texttt{pw} = \lambda \Phi \subseteq \{\texttt{S.w}, \texttt{F}.*\}.(\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{Socket.write};\ \texttt{f unit})$
- $\texttt{type}(\texttt{pw}) = \forall \Phi \subseteq \{\texttt{S.w}, \texttt{F}.*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi \cup \{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing$
- $\texttt{fx}(\texttt{type}(\texttt{pw})) = \{\texttt{S.w}\}$
- $\texttt{ho-safe}(\texttt{type}(\texttt{pw}), \{\texttt{S.w}\})$

## 8 Fixing Polywriter Incurs Effect

Imported capabilities:

- $\texttt{fw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{File.write}$
- $\texttt{pw} = \lambda \Phi \subseteq \{\texttt{File}.*\}.\ \texttt{Socket.write};\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{f unit})$
- $\varepsilon_s = \{\texttt{File.write}, \texttt{Socket.write}\}$

Types:

- $\texttt{type}(\texttt{fw}) = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{pw}) = \forall \Phi \subseteq \{\texttt{F}.*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \{\texttt{S.w}\}$

Effects

- $\texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{F.w}\}$
- $\texttt{effects}(\texttt{type}(\texttt{pw}))$
  $= \{\texttt{S.w}\} \cup \texttt{effects}((\texttt{Unit} \to_{\varnothing} \texttt{Unit}) \to_{\{\varnothing\}} \texttt{Unit})$
  $= \{\texttt{S.w}\}$
- $\texttt{effects}(\hat{\tau}_{\texttt{i}}) \subseteq \varepsilon_{\texttt{s}} = \texttt{True}$

Higher-Order Safety

- $\texttt{ho-safe}(\texttt{type}(\texttt{pw}), \varepsilon_s)$
  $= \texttt{ho-safe}(\forall \Phi \subseteq \{\texttt{F.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \{\texttt{S.w}\}, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \{\texttt{F.w}, \texttt{S.w}\} \not\subseteq \{\texttt{F.}*\}$

Fails for same reason as in previous example. To get it to typecheck, you need to include S.w in the upper bound of the polymorphic abstraction.

- $\texttt{pw} = \lambda\Phi \subseteq \{\texttt{S.w}, \texttt{F.}*\}.\ \texttt{Socket.write}; (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{f unit})$
- $\texttt{type}(\texttt{pw}) = \forall\Phi \subseteq \{\texttt{S.w}, \texttt{F.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \{\texttt{S.w}\}$
- $\texttt{fx}(\texttt{type}(\texttt{pw})) = \{\texttt{S.w}\}$
- $\texttt{ho-safe}(\texttt{type}(\texttt{pw}), \{\texttt{S.w}\})$

## 9 Instantiate 1 Poly and Pass To Another Poly

Imported capabilities. Note that the upper-bound on pw needs to have at least S.w, or it won't pass the higher-order safety check.

- $\texttt{fw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{File.write}$
- $\texttt{pw} = \lambda\Phi \subseteq \{\texttt{F.}*, \texttt{S.}*\}.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{f unit})$
- $\texttt{pa} = \lambda\Phi \subseteq \{\texttt{F.}*, \texttt{S.}*\}.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{S.w};\ \texttt{f unit})$
- $\varepsilon_s = \{\texttt{File.write}, \texttt{Socket.write}\}$

Types:

- $\texttt{type}(\texttt{fw}) = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{pw}) = \forall\Phi \subseteq \{\texttt{F.}*, \texttt{S.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \varnothing$
- $\texttt{type}(\texttt{pa}) = \forall\Phi \subseteq \{\texttt{F.}*, \texttt{S.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi\cup\{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing$

Effects

- $\texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{F.w}\}$
- $\texttt{effects}(\texttt{type}(\texttt{pw})) = \varnothing$
- $\texttt{effects}(\texttt{type}(\texttt{pa})) = \{\texttt{S.w}\}$
- $\texttt{effects}(\hat{\tau}_{\texttt{i}}) \subseteq \varepsilon_{\texttt{s}} = \texttt{True}$

Higher-Order Safety

- $\texttt{ho-safe}(\texttt{type}(\texttt{pw}), \varepsilon_s)$
  $= \texttt{ho-safe}(\forall \Phi \subseteq \{\texttt{F.}*, \texttt{S.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi} \texttt{Unit})\ \texttt{caps}\ \varnothing, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \{\texttt{F.w}, \texttt{S.w}\} \subseteq \{\texttt{F.}*, \texttt{S.}*\} \wedge \texttt{ho-safe}((\texttt{Unit} \to_{\{\texttt{F.}*,\texttt{S.}*\}} \texttt{Unit}) \to_{\{\texttt{F.}*,\texttt{S.}*\}} \texttt{Unit}, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \texttt{safe}(\texttt{Unit} \to_{\{\texttt{F.}*,\texttt{S.}*\}} \texttt{Unit}, \{\texttt{F.w}, \texttt{S.w}\}) \wedge \texttt{ho-safe}(\texttt{Unit}, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \{\texttt{F.w}, \texttt{S.w}\} \subseteq \{\texttt{F.}*, \texttt{S.}*\}$
  $= \texttt{True}$
- $\texttt{ho-safe}(\texttt{type}(\texttt{pa}), \varepsilon_s)$
  $= \texttt{ho-safe}(\forall \Phi \subseteq \{\texttt{F.}*, \texttt{S.}*\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi\cup\{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \{\texttt{F.w}, \texttt{S.w}\} \subseteq \{\texttt{F.}*, \texttt{S.}*\} \wedge \texttt{ho-safe}((\texttt{Unit} \to_{\{\texttt{F.}*,\texttt{S.}*\}} \texttt{Unit}) \to_{\{\{\texttt{F.}*,\texttt{S.}*\}} \texttt{Unit}, \{\texttt{F.w}, \texttt{S.w}\})$
  $= \texttt{safe}(\texttt{Unit} \to_{\{\texttt{F.}*,\texttt{S.}*\}} \texttt{Unit}, \texttt{F.w}, \texttt{S.w})$
  $= \{\texttt{F.w}, \texttt{S.w}\} \subseteq \{\texttt{F.}*, \texttt{S.}*\}$

## 10 Regular Function Returns Poly

Imported capabilities:

- $\texttt{fw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{File.write}$
- $\texttt{g} = \lambda \texttt{x} : \texttt{Unit}.\ (\lambda \Phi \subseteq \{\texttt{F.*}, \texttt{S.w}\}.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{S.w};\ \texttt{f unit}))$

Types:

- $\texttt{type}(\texttt{fw}) = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{g}) = \texttt{Unit} \to_{\varnothing} (\forall \Phi \subseteq \{\texttt{F.*}, \texttt{S.w}\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi \cup \{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing)$

Effects

- $\texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{F.w}\}$
- $\texttt{effects}(\texttt{type}(\texttt{g}))$
  $= \texttt{effects}(\forall \Phi \subseteq \{\texttt{F.*}, \texttt{S.w}\}.((\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\Phi \cup \{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing)$
  $= \texttt{effects}((\texttt{Unit} \to_{\varnothing} \texttt{Unit}) \to_{\{\texttt{S.w}\}} \texttt{Unit})$
  $= \{\texttt{S.w}\}$

Higher-Order Safety

- $\texttt{ho-safe}(\texttt{g}, \{\texttt{F.w}, \texttt{S.w}\}) \wedge \texttt{ho-safe}(\texttt{fw}, \{\texttt{F.w}, \texttt{S.w}\})$

## 11 Type Polymorphism (Contrived Example)

- $\texttt{tp} = \lambda \texttt{G} <: \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}.\ (\lambda \texttt{g} : \texttt{G}.\ \texttt{g unit};\ \texttt{S.w})$
- $\texttt{type}(\texttt{tp}) = \forall \texttt{G} <: \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}.\ (\texttt{G} \to_{\{\texttt{S.w}\}} \texttt{Unit})\ \texttt{caps}\ \varnothing$

To compute the effects of a type-polymorphic abstraction, sum up the effects incurred in evaluating the abstraciton body, and the effects captured in the return type of the abstraction body when the lower bound on the type is passed in. So in this case, you would be assuming that $\texttt{X} = \texttt{Unit} \to_{\varnothing} \texttt{Unit}$.

For higher-order effects, you sum up the effects captured by the upper-bound, and the effects captured by the return type of the abstraction body when the upper bound of the type is passed in. so in this case, you would be assuming that $\texttt{X} = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$.

- $\texttt{effects}(\texttt{type}(\texttt{tp})) = \{\texttt{F.w}, \texttt{S.w}\}$
- $\texttt{ho-effects}(\texttt{type}(\texttt{tp})) = \{\texttt{F.w}\}$

But if you rewrite `tp` like this:

- $\texttt{tp} = \lambda \Phi \subseteq \{\texttt{F.w}\}.\ (\lambda \texttt{g} : \texttt{Unit} \to_{\Phi} \texttt{Unit}.\ \texttt{g unit};\ \texttt{S.w})$
- $\texttt{type}(\texttt{tp}) = \forall \Phi \subseteq \{\texttt{F.w}\}.(\texttt{Unit} \to_{\Phi} \texttt{Unit}) \to_{\{\texttt{S.w}\} \cup \Phi} \texttt{Unit}\ \texttt{caps}\ \varnothing$

Then you get a tighter approximation on $\texttt{effects}(\texttt{type}(\texttt{tp}))$:

- $\texttt{effects}(\texttt{type}(\texttt{tp})) = \{\texttt{S.w}\}$
- $\texttt{ho-effects}(\texttt{type}(\texttt{tp})) = \{\texttt{F.w}\}$

Since we only have functions and resource sets, we could omit type polymorphism and still have the same expresiveness. But if we extended the system with more types, such as $\mathbb{N}, \mathbb{Z}, \mathbb{C}$, then you would lose expresiveness because you can't e.g. define a polymorphic plus function with the effect polymorphism construct.

Another approach: when you have an abstraction over a fucntion and its effects, rewrite that as abstracting over a type variable which ranges over the function's effects. This would possibly have to be a source code transformation, as it changes the types.

## 12 Poly Which Takes a Poly

Imported capabilities:

- $\texttt{fw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{File.write}$
- $\texttt{sw} = \lambda \texttt{x} : \texttt{Unit}.\ \texttt{Socket.write}$
- $\texttt{pid} = \lambda \Phi_1.\ \lambda \texttt{A}.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi_1} \texttt{Unit}.\ (\lambda \texttt{a} : \texttt{A}.\ \texttt{f unit};\ \texttt{a}))$
- $\texttt{pp} = \lambda \texttt{P} <: \texttt{type}(\texttt{pid}).\ \lambda \Phi_2.\ (\lambda \texttt{f} : \texttt{Unit} \to_{\Phi_2} \texttt{Unit}.\ (\lambda \texttt{p} : \texttt{P}.\ \texttt{f unit};\ \texttt{p}))$

Types:

- $\texttt{type}(\texttt{fw}) = \texttt{Unit} \to_{\{\texttt{F.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{sw}) = \texttt{Unit} \to_{\{\texttt{S.w}\}} \texttt{Unit}$
- $\texttt{type}(\texttt{pid}) = \forall \Phi_1.\ \forall \texttt{A}.\ (\texttt{Unit} \to_{\Phi_1} \texttt{Unit}) \to_{\varnothing} (\texttt{A} \to_{\Phi_1} \texttt{A})$
- $\texttt{type}(\texttt{pp}) = \forall \texttt{P} <: \texttt{type}(\texttt{pid}).\ \forall \Phi_2.\ (\texttt{Unit} \to_{\Phi_2} \texttt{Unit}) \to_{\varnothing} (\texttt{P} \to_{\Phi_2} \texttt{P})$