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 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,
   $\texttt{x} : \texttt{Unit},\ \phi \subseteq \{\texttt{LF.w}, \texttt{S.w}\}, \texttt{f} : \texttt{Str} \rightarrow_\phi \texttt{Unit},\ \texttt{msg} : \texttt{Str}$
   $\vdash \texttt{f msg} : \texttt{Unit with } \phi$
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 MakeHTTPServer, as in MakeHTTPServer unit {Socket.write}, it would type as follows:

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

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

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.

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

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 $\texttt{ho-safe}(fw, \varepsilon_s)$.

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

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

```
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)

```
λφ₁ ⊆ { File.write, File.read }.
  λφ₂ ⊆ φ₁.
    λf: Unit →_φ₁ Unit.
      λg: Unit →_φ₂ Unit.
        let _ = f unit in g unit
```

7 Stress-Testing Two-Variable Version of Effects

The intuition behind $\texttt{fx}(\hat{\tau}_A, \bar{\hat{\tau}})$ is that we are computing the possible effects of an expression of type $\hat{\tau}_A$, when only the capabilities in $\bar{\hat{\tau}}$ are in scope. For example, consider the example of a function which abstracts over any function with effects on `File`:

```
let pw =
  λφ ⊆ {File.*}.
    λf: Unit →_φ Unit.
      f unit
in ...
```

Consider a function which only writes to a file:

```
let fw =
  λx: Unit. File.write
```

Then consider the following use of an import construct:

```
import(ε_s)
  x₁ = pw, x₂ = fw
in e
```

What is the smallest, correct ε_s? A conservative answer is to say $\{\texttt{File}.*\}$ — indeed, `pw` is allowed to have any of these effects, provided someone gives it to them. But in the context of e, the only effect which can be realised is $\{\texttt{File.write}\}$, so an even better answer would be ε_s. This is the idea behind the two-variable version of `effects` — it attempts to give an upper-bound on the effects something can have by considering the capabilities in scope.

Some terminology: for simplicity, let $\texttt{type}(\hat{e})$ be the type obtained by type-checking $\hat{e}$ in the smallest possible context. In most cases it should be obvious what this is.In pretty much every following example, that is going to be $\varnothing$.

7.1 1 Poly, 1 Non-Poly

Consider a context where only the following two capabilites are in scope:

1. $\texttt{pw} = \lambda\phi \subseteq \{\texttt{File}.*\}.\ \lambda\texttt{f} : \texttt{Unit} \rightarrow_\phi \texttt{Unit}.\ \texttt{f unit}$
2. $\texttt{fw} = \lambda\texttt{x} : \texttt{Unit}.\ \texttt{File.write}$

Their types are the following:

1. $\hat{\tau}_1 = \texttt{type}(\texttt{pw}) = \forall\phi \subseteq \{\texttt{File}.*\}.\ (\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\phi \texttt{Unit caps } \varnothing \texttt{ with } \varnothing$
2. $\hat{\tau}_2 = \texttt{type}(\texttt{fw}) = \texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit with } \varnothing$

Their conservative effect approximations are:

1. $\texttt{effects}(\texttt{type}(\texttt{pw})) = \{\texttt{File}.*\}$
2. $\texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{File.write}\}$

The two-variable version of `effect` gives the following better approximation for the polymorphic type $\texttt{type}(\texttt{pw})$:

1. $\texttt{effects}(\texttt{type}(\texttt{pw}), \{\texttt{type}(\texttt{fw})\}) = \texttt{effects}(\texttt{type}(\texttt{pw})) \cap \texttt{effects}(\texttt{type}(\texttt{fw})) = \{\texttt{File.write}\}$

Which is a correct, tighter upper bound.

7.2 2 Poly Imports

The two capabilities are in scope:

1. $\texttt{pw} = \lambda\phi \subseteq \{\texttt{File}.*\}.\ \lambda\texttt{f} : \texttt{Unit} \rightarrow_\phi \texttt{Unit}.\ \texttt{f unit}$
2. $\texttt{sw} = \lambda\phi \subseteq \{\texttt{Socket}.*\}.\ \lambda\texttt{f} : \texttt{Unit} \rightarrow_\phi \texttt{Unit}.\ \texttt{f unit}$

Their types are the following:

1. $\hat{\tau}_1 = \texttt{type}(\texttt{pw}) = \forall\phi \subseteq \{\texttt{File}.*\}.\ (\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\phi \texttt{Unit caps } \varnothing \texttt{ with } \varnothing$
2. $\hat{\tau}_2 = \texttt{type}(\texttt{sw}) = \forall\phi \subseteq \{\texttt{Socket}.*\}.\ (\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\phi \texttt{Unit caps } \varnothing \texttt{ with } \varnothing$

Which have the following conservative effect approximations:

1. $\texttt{effects}(\texttt{type}(\texttt{pw})) = \{\texttt{File}.*\}$
2. $\texttt{effects}(\texttt{type}(\texttt{sw})) = \{\texttt{Socket}.*\}$

The two-variable version of `effect` gives the following better approximations:

1. $\texttt{effects}(\texttt{type}(\texttt{pw}), \{\texttt{type}(\texttt{sw})\}) = \texttt{effects}(\texttt{type}(\texttt{pw})) \cap \texttt{effects}(\texttt{type}(\texttt{sw})) = \{\texttt{File}.*\} \cap \{\texttt{Socket}.*\} = \varnothing$
2. $\texttt{effects}(\texttt{type}(\texttt{sw}), \{\texttt{type}(\texttt{pw})\}) = \texttt{effects}(\texttt{type}(\texttt{sw})) \cap \texttt{effects}(\texttt{type}(\texttt{pw})) = \{\texttt{Socket}.*\} \cap \{\texttt{File}.*\} = \varnothing$

These upper bounds are tighter. They are also correct — no matter how you fix `pw` or `sw`, there is no way to pass one to the other in order to invoke an effect.

7.3 1 Poly, 1 Unusable Non-Poly

The following two capabilities are in scope. This time, fw writes to both File and Socket.

1. $\texttt{pw} = \lambda\phi \subseteq \{\texttt{File}.*\}.\ \lambda\texttt{f} : \texttt{Unit} \rightarrow_{\phi} \texttt{Unit}.\ \texttt{f unit}$
2. $\texttt{fw} = \lambda\texttt{x} : \texttt{Unit}.\ \texttt{File.write};\ \texttt{Socket.write}$

Their types are the following:

1. $\hat{\tau}_1 = \texttt{type(pw)} = \forall\phi \subseteq \{\texttt{File}.*\}.\ (\texttt{Unit} \rightarrow_{\phi} \texttt{Unit}) \rightarrow_{\phi} \texttt{Unit caps } \varnothing \texttt{ with } \varnothing$
2. $\hat{\tau}_2 = \texttt{type(fw)} = \texttt{Unit} \rightarrow_{\{\texttt{File.write},\texttt{Socket.write}\}} \texttt{Unit with } \varnothing$

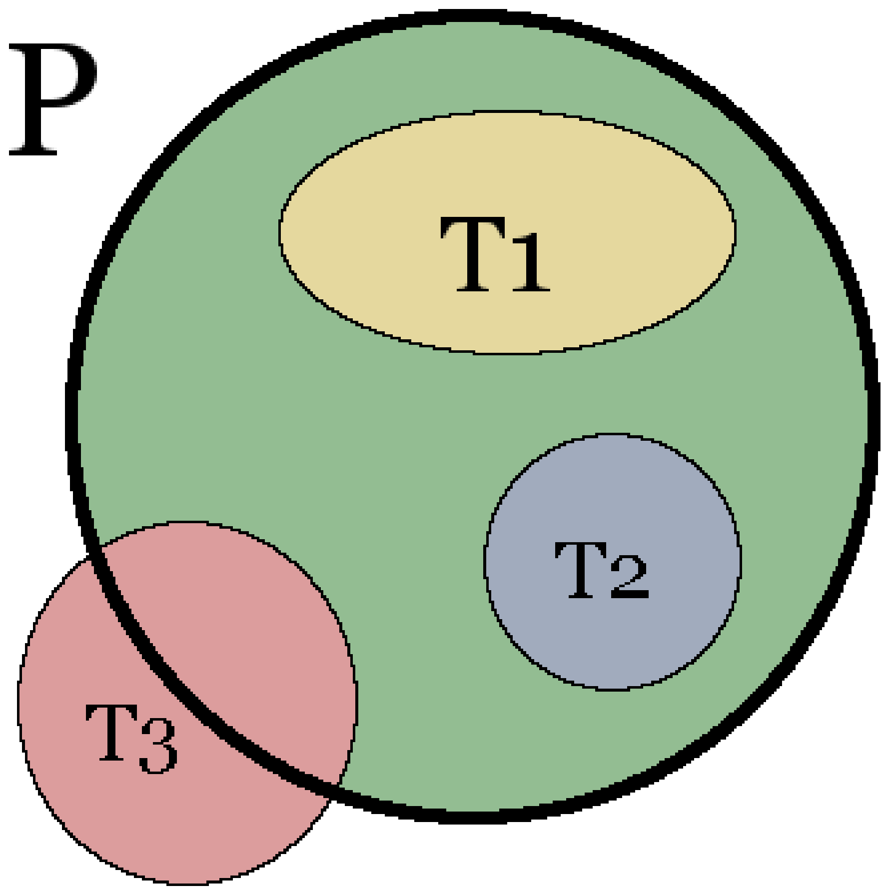
Their conservative effect approximations are:

1. $\texttt{effects(type(pw))} = \{\texttt{File}.*\}$
2. $\texttt{effects(type(fw))} = \{\texttt{File.write}, \texttt{Socket.write}\}$

The two-variable version of effects will give the following approximation for type(pw):

1. $\texttt{effects(type(pw)}, \{\texttt{type(fw)}\}) = \{\texttt{File}.*\} \cap \{\texttt{File.write}, \texttt{Socket.write}\} = \{\texttt{File.write}\}$

This is a better approximation than File.$*$, but it's still not a tight approximation. The tight approximation in this situation is $\varnothing$, because you can never pass fw to any instantiation of pw — any instantiation of pw can only be passed a function with effects on File, but fw has effects on Socket. The issue: we only intersect the polymorphic effects with the effects of a capability in scope if that capability's effects are contained in the upper-bound of the polymorphic effects. If not, the maximal set of effects that could be incurred with that capability is $\varnothing$, because it can't be passed to the polymorphic code. The following diagram illustrates:



The circles represent the effects of the labelled types. So the green circle is $\texttt{effects}(P)$. We currently approximate $\texttt{effects}(p, \{T_1, T_2, T_3\})$ as $\texttt{effects}(p) \cap (\texttt{effects}(T_1) \cup \texttt{effects}(T_2) \cup \texttt{effects}(T_3))$. This approximation includes those effects in both $\texttt{effects}(P) \cap \texttt{effects}(T_3)$ — but these effects can't ever be used by p, since the capability T_3 contains more effects than stipulated by the upper-bound of p. Therefore we want to exclude T_3 from our union, and instead give the approximation as $\texttt{effects}(p) \cap (\texttt{effects}(T_1) \cup \texttt{effects}(T_2))$. That is, we want to exclude $\texttt{effects}(T_3)$ from our approximation because it contains effects outside of $\texttt{effects}(P)$.

Here are two proposed amendments to $\texttt{effects}(p, \overline{\tau})$:

1. Have two cases based on whether $\texttt{effects}(\hat{\tau}_i) \subseteq \texttt{effects}(p)$. They might look like the following

$$\texttt{effects}(p, \{\hat{\tau}_i\}) = \bigcup_i \begin{cases} \texttt{effects}(\hat{\tau}_i) \cap \texttt{effects}(p) & \text{if } \texttt{effects}(\hat{\tau}_i) \subseteq \texttt{effects}(P) \\ \varnothing & \text{otherwise} \end{cases}$$

2. Notice that the result is always going to be a subset of $\texttt{effects}(p)$ (either $\varnothing$ or the intersection of $\texttt{effects}(p)$ with the effects of some capability in scope).

$$\texttt{effects}(p, \{\hat{\tau}_i\}) = \bigcup [\mathcal{P}(p) \cap \bigcup_i \{\texttt{effects}(\hat{\tau}_i)\}]$$

   Now if there is a capability in $\texttt{effects}(\hat{\tau}_i) \setminus \texttt{effects}(p)$ then $\{\texttt{effects}(\hat{\tau}_i)\}$ won't be in $\mathcal{P}(p)$, so the result will be $\varnothing$.

7.4 Instantiate 1 Poly and Pass To Another Poly

Consider the following capabilities:

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

pa = λφ ⊆ {File.*}.
       λf: Unit →φ Unit.
          let _ = Socket.write in f unit

fw = λx: Unit. File.write
```

Note how pa, when fixed with an effect-set, will ask for a function with those effects and then instrument it with the Socket.write effect.

The maximal set of effects that can be achieved with these capabilities is $\{File.write, Socket.write\}$, in the following way:

```
import({File.write, Socket.write})
   pw = pw, fw = fw, pa = pa
in
   let pw2 = pw {File.write, Socket.write} in
   let pa2 = pw {File.write} in
   pw2 (pa2 fw) /* incurs File.write, Socket.write */
```

Their types are the following:

1. $\texttt{type(pw)} = \forall\phi \subseteq \{\texttt{File.}*, \texttt{Socket.}*\}.\ (\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\phi \texttt{Unit caps } \varnothing \texttt{ with } \varnothing$
2. $\texttt{type(pa)} = \forall\phi \subseteq \{\texttt{File.}*\}.\ (\texttt{Unit} \rightarrow_\phi \texttt{Unit}) \rightarrow_\varnothing (\texttt{Unit} \rightarrow_{\phi \cup \{\texttt{Socket.write}\}} \texttt{Unit})$
3. $\texttt{type(fw)} = \texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit with } \varnothing$

Their conservative effect approximations are:

1. $\texttt{effects(type(pw))} = \{\texttt{File.}*, \texttt{Socket.}*\}$
2. $\texttt{effects(type(pa))} = \{\texttt{File.}*, \texttt{Socket.write}\}$
3. $\texttt{effects(type(fw))} = \{\texttt{File.write}\}$

If we use the two-variable version of effects to approximate pa, then we get the following:

1. $\texttt{effects}(\texttt{type(pw)}, \{\texttt{type(pa)}, \texttt{type(fw)}\}) = \{\texttt{File}.*, \texttt{Socket.write}\}$

Which is correct. It also doesn't matter whether you use the old version of `effects` or the updated version from the previous section; they give the same answer. However, if you try to apply the two-variable version of `effects` to approximate `pw` you get:

2. $\texttt{effects}(\texttt{type(pw)}, \{\texttt{type(pa)}, \texttt{type(fw)}\}) = \{\texttt{File}.*, \texttt{Socket.write}\}$

Which is correct, but not a tight upper-bound. Both versions of `effects` give this answer. The problem arises from when you intersect the (conservative) effects of `pw` with the (conservative) effects of `pa`. Both $\texttt{effects}(pa)$ and $\texttt{effects}(pw)$ have operations on `File` which can't ever be invoked, so when their conservative approximations are intersected, we get every operation on `File` in the result.

7.5 Pass Uninstantiated Poly To Another Poly

Consider the following capabilities:

```
/* A polymorphic id function over a type A, which incurs an effect φ_1 before returning the input argument */
pid = λΦ_1.λA.
   λf: Unit→_{Φ_1}Unit.
      λa: A. let _ = f unit in a

/* A polymorphic function which takes a polymorphic abstraction P and wraps it in a computation f with effect Φ_3 */
pp = λP <: ∀Φ_2.∀A.(Unit →_{Φ_2} Unit) →_∅ A →_{Φ_2} A
   λΦ_3. λf : Unit →_{Φ_3} Unit. λp : P.
      let _ = f unit in p

/* Capabilities which capture the write operation on File and Socket */
fw = λx:Unit. File.write
sw = λx:Unit. Socket.write
```

They have the following conservative effect approximations:

1. $\texttt{effects}(pp) = \texttt{effects}(pid) = R \times \Pi$
2. $\texttt{effects}(fw) = \{\texttt{File.write}\}$
3. $\texttt{effects}(sw) = \{\texttt{Socket.write}\}$

The naive definition does not give a better approximation. For example,

$\texttt{effects}(pid, pp, fw, sw) =$
$(\texttt{effects}(pid) \cap \texttt{effects}(pp)) \cup (\texttt{effects}(pid) \cap \texttt{effects}(fw)) \cup (\texttt{effects}(pid) \cap \texttt{effects}(sw)) =$
$R \times \Pi \cup \{\texttt{File.write}\} \cup \{\texttt{Socket.write}\} =$
$R \times \Pi$