

1 Basic Effect Polymorphism

Pseudo-Wyvern

```

1 def polymorphicWriter(x: T <: {File, Socket}): Unit with T.write =
2   x.write
3
4   /* below invocation should typecheck with File.write as its only effect */
5   polymorphicWriter File

```

λ -Calculus

```

1 let pw =  $\lambda\phi \subseteq \{\text{File.write}, \text{Socket.write}\}.$ 
2    $\lambda f: \text{Unit} \rightarrow_{\phi} \text{Unit}.$ 
3     f unit
4
5 in let makeWriter =  $\lambda r: \{\text{File}, \text{Socket}\}.$ 
6    $\lambda x: \text{Unit}.$  r.write
7
8 in (pw {File.write}) (makeWriter File)

```

Typing

To type the definition of `polymorphicWriter`:

1. By ε -APP
 $\phi \subseteq \{\text{F.w}, \text{S.w}\}, x: \text{Unit} \rightarrow_{\phi} \text{Unit} \vdash x \text{ unit} : \text{Unit with } \phi.$
2. By ε -ABS
 $\phi \subseteq \{\text{F.w}, \text{S.w}\} \vdash \lambda x: \text{Unit} \rightarrow_{\phi} \text{Unit}. x \text{ unit} : (\text{Unit} \rightarrow_{\phi} \text{Unit}) \rightarrow_{\phi} \text{Unit with } \emptyset$
3. By ε -POLYFXABS,
 $\vdash \forall \phi \subseteq \{\text{S.w}, \text{F.w}\}. \lambda x: \text{Unit} \rightarrow_{\phi} \text{Unit}. x \text{ unit} : \forall \phi \subseteq \{\text{F.w}, \text{S.w}\}. (\text{Unit} \rightarrow_{\phi} \text{Unit}) \rightarrow_{\phi} \text{Unit caps } \emptyset \text{ with } \emptyset$

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

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

The judgement can be simplified to:

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

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.

```

1 module HTTPServer
2
3 def init(out: A <: {File, Socket}): Str  $\rightarrow_{A.write}$  Unit with  $\emptyset$  =
4    $\lambda \text{msg}: \text{Str}.$ 
5     if (msg == "POST") then out.write("post response")
6     else if (msg == "GET") then out.write("get response")
7     else out.write("client error 400")

```

The main module calls `HTTPServer.init` with the `Socket` it should be writing to.

```

1 module Main
2   require HTTPServer, Socket
3
4   def main(): Unit =
5     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.

```

1 module Testing
2   require HTTPServer, LogFile
3
4   def testSocket(): =
5     HTTPServer.init(LogFile) "GET /index.html"

```

λ -Calculus

The `HTTPServer` module:

```

1 MakeHTTPServer =  $\lambda x$ : Unit.
2    $\lambda \phi \subseteq \{\text{LogFile.write}, \text{Socket.write}\}.$ 
3      $\lambda f$ :  $\text{Str} \rightarrow_{\phi} \text{Unit}.$ 
4        $\lambda \text{msg}$ :  $\text{Str}.$ 
5         f msg

```

The `Main` module:

```

1 MakeMain =  $\lambda \text{hs}$ : HTTPServer.  $\lambda \text{sock}$ :  $\{\text{Socket}\}.$ 
2    $\lambda x$ : Unit.
3     let socketWriter = ( $\lambda s$ :  $\{\text{Socket}\}.$   $\lambda x$ : Unit. s.write) sock in
4     let theServer = hs {Socket.write} socketWriter in
5     theServer "GET/index.html"

```

The `Testing` module:

```

1 MakeTest =  $\lambda \text{hs}$ : HTTPServer.  $\lambda lf$ :  $\{\text{LogFile}\}.$ 
2    $\lambda x$ : Unit.
3     let logFileWriter = ( $\lambda l$ :  $\{\text{LogFile}\}.$   $\lambda x$ : Unit. l.write) lf in
4     let theServer = hs {LogFile.write} logFileWriter in
5     theServer "GET/index.html"

```

A single, desugared program for production would be:

```

1 let MakeHTTPServer =  $\lambda x$ : Unit.
2    $\lambda \phi \subseteq \{\text{LogFile.write}, \text{Socket.write}\}.$ 
3      $\lambda f$ :  $\text{Str} \rightarrow_{\phi} \text{Unit}.$ 
4        $\lambda \text{msg}$ :  $\text{Str}.$ 
5         f msg
6
7 in let Run =  $\lambda \text{Socket}$ :  $\{\text{Socket}\}.$ 
8   let HTTPServer = MakeHTTPServer unit in
9   let Main = MakeMain HTTPServer Socket in
10  Main unit
11
12 in Run Socket

```

A single, desugared program for testing would be:

```

1 let MakeHTTPServer =  $\lambda x$ : Unit.
2    $\lambda \phi \subseteq \{\text{LogFile.write}, \text{Socket.write}\}.$ 
3      $\lambda f$ :  $\text{Str} \rightarrow_{\phi} \text{Unit}.$ 
4        $\lambda \text{msg}$ :  $\text{Str}.$ 
5         f msg
6

```

```

7  in let Run = λLogFile: {LogFile}.
8    let HTTPServer = MakeHTTPServer unit in
9    let Main = MakeMain HTTPServer LogFile in
10   Main unit
11
12  in Run LogFile

```

Note how the HTTPServer code is identical in the testing and production examples.

Typing

```

1  let MakeHTTPServer = λx: Unit.
2    λφ ⊆ {LogFile.write, Socket.write}.
3      λf: Str →φ Unit.
4        λmsg: Str.
5          f msg

```

To type MakeHTTPServer:

1. By ε -APP,
 $x : \text{Unit}, \phi \subseteq \{\text{LF.w}, \text{S.w}\}, f : \text{Str} \rightarrow_{\phi} \text{Unit}, \text{msg} : \text{Str}$
 $\vdash f \text{ msg} : \text{Unit} \text{ with } \phi$
2. By ε -ABS,
 $x : \text{Unit}, \phi \subseteq \{\text{LF.w}, \text{S.w}\}, f : \text{Str} \rightarrow_{\phi} \text{Unit}$
 $\vdash \lambda \text{msg} : \text{Str}. f \text{ msg} : \text{Str} \rightarrow_{\phi} \text{Unit} \text{ with } \emptyset$
3. By ε -ABS,
 $x : \text{Unit}, \phi \subseteq \{\text{LF.w}, \text{S.w}\}$
 $\vdash \lambda f : \text{Str} \rightarrow_{\phi} \text{Unit}. \lambda \text{msg} : \text{Str}. f \text{ msg} :$
 $(\text{Str} \rightarrow_{\phi} \text{Unit}) \rightarrow_{\emptyset} (\text{Str} \rightarrow_{\phi} \text{Unit}) \text{ with } \emptyset$
4. By ε -POLYFXABS,
 $x : \text{Unit}$
 $\vdash \lambda \phi \subseteq \{\text{LF.w}, \text{S.w}\}. \lambda f : \text{Str} \rightarrow_{\phi} \text{Unit}. \lambda \text{msg} : \text{Str}. f \text{ msg} :$
 $\forall \phi \subseteq \{\text{LF.w}, \text{S.w}\}. (\text{Str} \rightarrow_{\phi} \text{Unit}) \rightarrow_{\emptyset} (\text{Str} \rightarrow_{\phi} \text{Unit}) \text{ caps } \emptyset \text{ with } \emptyset$
5. By ε -ABS,
 $\vdash \lambda x : \text{Unit}. \lambda \phi \subseteq \{\text{LF.w}, \text{S.w}\}. \lambda f : \text{Str} \rightarrow_{\phi} \text{Unit}. \lambda \text{msg} : \text{Str}. f \text{ msg} :$
 $\text{Unit} \rightarrow_{\emptyset} \forall \phi \subseteq \{\text{LF.w}, \text{S.w}\}. (\text{Str} \rightarrow_{\phi} \text{Unit}) \rightarrow_{\emptyset} (\text{Str} \rightarrow_{\phi} \text{Unit}) \text{ caps } \emptyset \text{ with } \emptyset$

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

6. By ε -POLYFXAPP,
 $x : \text{Unit}$
 $\vdash \text{MakeHTTPServer unit } \{\text{S.w}\} :$
 $(\text{Str} \rightarrow_{\{\text{S.w}\}} \text{Unit}) \rightarrow_{\emptyset} (\text{Str} \rightarrow_{\{\text{S.w}\}} \text{Unit}) \text{ with } \emptyset$

After fixing the polymorphic set of effects, possessing this function only gives you access to the `Socket.write` effect.

3 Map Function

Pseudo-Wyvern

```

1  def map(f: A →φ B, l: List[A]): List[B] with φ =
2    if isnil l then []
3    else cons (f (head l)) (map (tail l f))

```

λ-Calculus

```

1  map = λφ. λA. λB.
2    λf: A→φB.
3    (fix (λmap: List[A] → List[B])).
4    λl: List[A].
5      if isnil l then []
6      else cons (f (head l)) (map (tail l f)))

```

Typing

- This has the type: $\forall\phi.\forall A.\forall B.(A \rightarrow_{\phi} B) \rightarrow_{\emptyset} \text{List}[A] \rightarrow_{\phi} \text{List}[B]$ with \emptyset .
- `map \emptyset` 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

```

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

```

1  import({File.*})
2    pw = polywriter
3    f = File
4  in
5    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 $\text{effects}(pw) = \text{ho-effects}(pw) = \{\text{File.write}, \text{Socket.write}\}$. Then:

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

And also:

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

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

```

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

```

The last line is not true, because $\{\text{File.*}\} \subseteq \{\text{File.write}, \text{Socket.write}\}$ is not true. The intuition 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 =  $\lambda \phi \subseteq \{\text{File.write}, \text{Socket.write}\}. \lambda f: \text{Unit} \rightarrow_\phi \text{Unit}. f \text{ unit}$ 
2
3 let fwriter =  $\lambda x: \text{Unit}. \text{File.write}$ 
4
5 import( $\{\text{File.write}\}$ )
6   pw = polywriter
7   fw = fwriter
8 in
9   pw  $\{\text{File.write}\}$  fw

```

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

```

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

```

And also

```

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

```

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

```

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

```

```

ho-safe( $fw, \varepsilon_s$ )
= ho-safe( $\text{Unit} \rightarrow_{\{\text{File.write}\}} \text{Unit}, \{\text{File.write}\})$ )
=  $\text{safe}(\text{Unit}, \{\text{File.write}\}) \wedge \text{ho-safe}(\text{Unit}, \{\text{File.write}\})$ 
= 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).

$$\begin{aligned}
& \text{ho-safe}((\text{Unit} \rightarrow_{\{\text{File.write}\}} \text{Unit}) \rightarrow_{\{\text{File.write}\}} \text{Unit}, \{\text{File.*}, \text{Socket.*}\}) \\
&= \text{safe}(\text{Unit} \rightarrow_{\{\text{File.write}\}} \text{Unit}, \{\text{File.*}, \text{Socket.*}\}) \wedge \text{ho-safe}(\text{Unit}, \{\text{File.*}, \text{Socket.*}\}) \\
&= \text{safe}(\text{Unit} \rightarrow_{\{\text{File.write}\}} \text{Unit}, \{\text{File.*}, \text{Socket.*}\}) \\
&= \{\text{File.*}, \text{Socket.*}\} \subseteq \{\text{File.*}\}
\end{aligned}$$

which is false.

6 Composing polymorphic functions (artificial example)

```

1 λφ1 ⊆ { File.write, File.read }.
2   λφ2 ⊆ φ1.
3     λf: Unit →φ1 Unit.
4       λg: Unit →φ2 Unit.
5         let _ = f unit in g unit

```

7 Stress-Testing Two-Variable Version of Effects

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

```

1 let pw =
2   λφ ⊆ {File.*}.
3     λf: Unit →φ Unit.
4       f unit
5 in ...

```

Consider a function which only writes to a file:

```

1 let fw =
2   λx: Unit. File.write

```

Then consider the following use of an import construct:

```

1 import(εs)
2   x1 = pw, x2 = fw
3 in e

```

What is the smallest, correct ε_s ? A conservative answer is to say $\{\text{File.*}\}$ — 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 $\{\text{File.write}\}$, 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 $\text{type}(\hat{e})$ 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 \emptyset .

7.1 1 Poly, 1 Non-Poly

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

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

Their types are the following:

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

Their conservative effect approximations are:

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

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

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

Which is a correct, tighter upper bound.

7.2 2 Poly Imports

The two capabilities are in scope:

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

Their types are the following:

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

Which have the following conservative effect approximations:

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

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

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

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

Their types are the following:

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

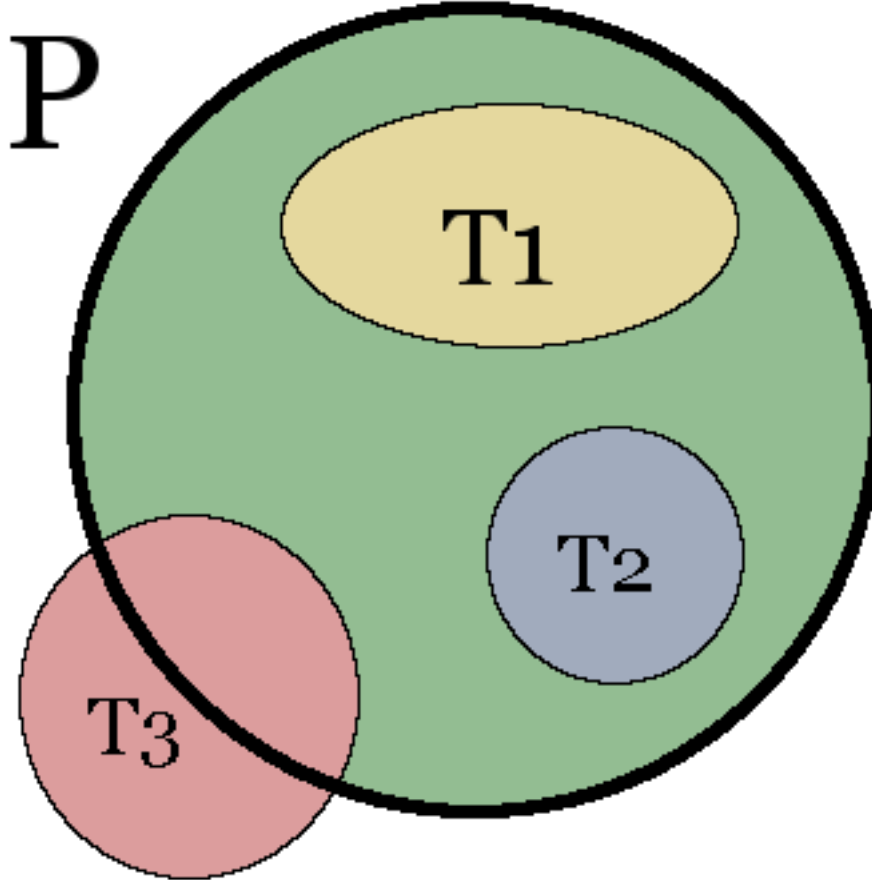
Their conservative effect approximations are:

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

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

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

This is a better approximation than `File.*`, but it's still not a tight approximation. The tight approximation in this situation is \emptyset , 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 \emptyset , 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 $\text{effects}(P)$. We currently approximate $\text{effects}(p, \{T_1, T_2, T_3\})$ as $\text{effects}(p) \cap (\text{effects}(T_1) \cup \text{effects}(T_2) \cup \text{effects}(T_3))$. This approximation includes those effects in both $\text{effects}(P) \cap \text{effects}(T_3)$ — 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 $\text{effects}(p) \cap (\text{effects}(T_1) \cup \text{effects}(T_2))$. That is, we want to exclude $\text{effects}(T_3)$ from our approximation because it contains effects outside of $\text{effects}(P)$.

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

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

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

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

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

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

7.4 Instantiate 1 Poly and Pass To Another Poly

Consider the following capabilities:

```

1 pw = λφ ⊆ {File.*, Socket.*}.
2   λf: Unit →φ Unit. f unit
3
4 pa = λφ ⊆ {File.*}.
5   λf: Unit →φ Unit.
6     let _ = Socket.write in f unit
7
8 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:

```

1 import({File.write, Socket.write})
2 pw = pw, fw = fw, pa = pa
3 in
4   let pw2 = pw {File.write, Socket.write} in
5   let pa2 = pw {File.write} in
6   pw2 (pa2 fw) /* incurs File.write, Socket.write */
```

Their types are the following:

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

Their conservative effect approximations are:

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

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

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

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

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 $\text{effects}(\text{pa})$ and $\text{effects}(\text{pw})$ 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:

```

1  /* A polymorphic id function over a type A, which incurs an effect  $\phi_1$  before returning the input argument */
2  pid =  $\lambda\Phi_1.\lambda A.$ 
3       $\lambda f:\text{Unit} \rightarrow_{\Phi_1} \text{Unit}.$ 
4           $\lambda a:A. \text{let } _ = f \text{ unit in } a$ 
5
6  /* A polymorphic function which takes a polymorphic abstraction P and wraps it in a computation f with effect  $\Phi_3$  */
7  pp =  $\lambda P <: \forall \Phi_2. \forall A. (\text{Unit} \rightarrow_{\Phi_2} \text{Unit}) \rightarrow_{\emptyset} A \rightarrow_{\Phi_2} A$ 
8       $\lambda \Phi_3. \lambda f:\text{Unit} \rightarrow_{\Phi_3} \text{Unit}. \lambda p:P.$ 
9           $\text{let } _ = f \text{ unit in } p$ 
10
11 /* Capabilities which capture the write operation on File and Socket */
12 fw =  $\lambda x:\text{Unit}. \text{File.write}$ 
13 sw =  $\lambda x:\text{Unit}. \text{Socket.write}$ 

```

They have the following conservative effect approximations:

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

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

$$\begin{aligned}
 \text{effects}(\text{pid}, \text{pp}, \text{fw}, \text{sw}) &= \\
 (\text{effects}(\text{pid}) \cap \text{effects}(\text{pp})) \cup (\text{effects}(\text{pid}) \cap \text{effects}(\text{fw})) \cup (\text{effects}(\text{pid}) \cap \text{effects}(\text{sw})) &= \\
 R \times \Pi \cup \{\text{File.write}\} \cup \{\text{Socket.write}\} &= \\
 R \times \Pi
 \end{aligned}$$

7.6 Regular Function Returns Poly

Consider the following capabilities:

```

1  fw =  $\lambda x:\text{Unit}. \text{File.write}$ 
2
3  sw =  $\lambda x:\text{Unit}. \text{Socket.write}$ 
4
5  pw =  $\lambda \Phi \subseteq \{\text{File}.*, \text{Socket}.*\}.$ 
6       $\lambda f:\text{Unit} \rightarrow_{\Phi} \text{Unit}. f \text{ unit}$ 
7
8  g =  $\lambda x:\text{Unit}. \text{pw}$ 

```

Now consider the following **import** expression, which only imports **fw**, **sw**, and **g**.

```

1  import({File.write, Socket.write})
2  fw = fw, sw = sw, g = g
3  in ...

```

The types are the following:

1. $\text{type}(\text{fw}) = \text{Unit} \rightarrow_{\{\text{File.write}\}} \text{Unit}$
2. $\text{type}(\text{sw}) = \text{Unit} \rightarrow_{\{\text{Socket.write}\}} \text{Unit}$
3. $\text{type}(\text{pw}) = \lambda\Phi \subseteq \{\text{File.*}, \text{Socket.*}\}. (\text{Unit} \rightarrow_{\Phi} \text{Unit}) \rightarrow_{\Phi} \text{Unit}$
4. $\text{type}(\text{g}) = \text{Unit} \rightarrow_{\emptyset} \text{type}(\text{pw})$

Their conservative effects:

1. $\text{effects}(\text{type}(\text{fw})) = \{\text{File.write}\}$
2. $\text{effects}(\text{type}(\text{sw})) = \{\text{Socket.write}\}$
3. $\text{effects}(\text{type}(\text{pw})) = \{\text{File.*}, \text{Socket.*}\}$
4. $\text{effects}(\text{type}(\text{g})) = \{\text{File.*}, \text{Socket.*}\}$

Note that since g is not polymorphic, any of the approximations of its effects so far will give $\{\text{File.*}, \text{Socket.*}\}$.