Capabilities for Effects

A/Prof Alex Potanin

Formalism by Aaron Craig as Undergraduate Thesis student at VUW in 2017/2018

CAPABILITY-FLAVOURED EFFECTS

Capability Safety

- Capability-safe languages prohibit ambient authority
  - All authority derives from previous authority, starting at the entry point of the program
  - A component can't exercise authority unless you give it a capability to do so
- Can be used to quantify risk of executing code [Drossopolou] and ensure least privilege [Saltzer]
- Do capabilities help existing formal reasoning techniques, such as effects?

Effects

- Describe "intensional information" about how a program executes (Neilson & Nelson)
  - *Int → Int* (unannotated function type)
  - *Int –{File.write}→ Int* (annotated function type)
- Limited mainstream use; too verbose? (Rytz)
- Inference helps reduce verbosity
  - Need to analyse source code
  - Back to manual reasoning if it fails

Capability-Flavoured Effects

- In a capability-safe setting, any effect on a resource must happen through a capability
- By tracking capabilities, we also track effects
- What can we say at the boundary where annotated code passes capabilities into unannotated code?

- ```
  import(File.append)
    logger: String –{File.append}→ Unit
  in
    e // arbitrary, unannotated code
  ```
  ```

Capability-Flavoured Effects

- Can safely determine effects of unannotated code by inspecting the capabilities we give it
  - Only have to inspect its type, not its source code
- Effect-conscious capability-safe code can reason about what untrusted, capability-safe code will do
- Our work: formulates a minimal, sound lambda calculus and type system to demonstrate this

Imports

```
import(File.append)
  log: String -{File.append}→ Unit
in
  log("doing some logging")
```

- Pass in capabilities, execute unannotated code
- Unannotated code must type with *exactly* the free variables imported
- Programmer *selects* authority as `{File.*}`
- Statically: accept/reject, if `{File.*}` is a safe upper-bound on effects

Multiple Imports

```
import(File.*)
  makeFile: Unit -{File.create}→ Unit
  pureApply: (Unit -∅→ Unit) -∅→ Unit
in
  pureApply(makeFile)
```

- Input to `pureApply` has same type as `makeFile` (modulo effect annotations)
- Don't want `pureApply` to violate its annotation by incurring a `File.create` effect in the unannotated code
- Need to ensure all imports are allowed the selected authority before passing them in

Higher-Order Effects

- Regular effect: you possess capability for effect
- Higher-order effect: allowed to incur effect, but you need to be given the capability

```
… // some omitted set up code
def log(msg: String, sock: Socket) =
   file.append(“hello”)
   sock.append(“they're logging”)
```

- Assuming this typechecks…
  - `File.append` is a regular effect
  - `Socket.append` is higher order

Higher-Order Safety

- An annotated type τ is *higher-order safe* for a set of effects ε if $\varepsilon \subseteq$ `ho-effects(`τ`)`
- Intuitively: an expression of type τ must be allowed to incur the effects in ε
- To safely check an import, all imports must be higher-order safe for the selected authority

```
import(File.create)
  makeFile: Unit –{File.create}→ Unit
  pureApply: (Unit –∅→ Unit) –∅→ Unit
in
  pureApply(log)
```

Return Types

- Unannotated code might return a function/capability
- Need to annotate it with effects to safely effect-check rest of annotated code

```
let result =
  import(File.*)
    f: {File}
  in
    def tricky(): Unit =
      f.write("hello")
result()
```

- The type of tricky is `Unit → Unit`, which annotates as `Unit –{File.*}→ Unit`

Return Types

- Unannotated code might return a function that can later be used elsewhere in the annotated world
- Need to understand what effects it has to safely effect-check annotated code using it

```
import(File.*)
  f: {File}
in
  def tricky(): Unit =
    f.write("hello")
```

- The type of tricky is `Unit → Unit`, which annotates as `Unit –{File.*}→ Unit`

Returning Higher-Order Effects

```
import(File.*)
  f: {File}
in
  def myFunc(msg: String, s: Socket): Unit =
    s.write("they're logging")
    f.write(msg)
```

- Safe to execute this code
- Unsafe to annotate return type with `{File.*}`
- Must make sure return type doesn't ask for a capability (`Socket`) whose effects haven't been selected
  - This example rejects because `String → Socket → Unit` has the higher-order effects `{Socket.*}`

Polymorphic Types

- Polymorphic types let you write type-generic code
- Polymorphic effects let you write effect-generic code

```
// Define a new effect to simplify function definition
effect write = {File.write, Socket.write}

// Takes a write function, uses it to write a message, logs
def writeData<φ ⊆ write>(s: String, write: String –φ→ Unit) =
  write(s)
  file.append(“wrote to writer”)

type WriteDataFunc = typeof(writeData)
```

Polymorphic Imports

```
effect write = {File.write, Socket.write}
import(File.append, File.write, Socket.write)
  writeData: WriteDataFunc<write>
  fwriter: String –{File.write, File.append}→ Unit
in
  e
```

- Can approximate effects of `writeData` with its polymorphic upper bound `{File.write, Socket.write}`
- Can approximate effects of the unannotated code as `{File.append, File.write, Socket.write}`

Polymorphic Imports

- Lots of generic code doesn't have an upper bound on its possible effects
  - map, fold/reduce, filter, zip, collections
- To incur an effect with generics you must instantiate with something concrete that can invoke the effect
  - Capability for the effect must have been imported
- Can tighten the upper-bound by looking at other capabilities that have been imported

Polymorphic Imports

```
effect write = {File.write, Socket.write}
import(File.append, File.write, Socket.write)
  writeData: WriteDataFunc<write>
  fwriter: String –{File.write, File.append}→ Unit
in
  e
```

- Nothing imported can incur `Socket.write` so we can ignore that as a possibility for `writeData`
- Upper-bound on `writeData` tightens to `{File.write}`
- Better approximation of effects of *e* is `{File.write, File.append}`

Overall

- Capability-safe design enables reasoning at module boundaries about the effects of unannotated code
- Must restrict capabilities passed in based on their higher-order effects
- Finer reasoning needed for useful polymorphics

Paper by Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich at ICFEM 2018 conference

CAPABILITIES: EFFECTS FOR FREE

CAPITAL THINKING. GLOBALLY MINDED.
MAI I TE IHO KI TE PAE

VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA
NEW ZEALAND

Motivation

- Consider a program which calls a logger component:

```
module def logger(f:{File}):Logger
def log(x: String): Unit
```

```
module def client(logger: Logger)
def run(): Unit = logger.log(x)
```

  - We pass the logger a file expecting it to append to it.
  - But how do we ensure that is all it does?

- In Java, once the logger has the file, it can do anything it wants: “ambient authority”.

- Capabilites have been used informally to reason about resource use — can we use them formally?

Approach

- We look at adding capability-based reasoning to a formal system for reasoning about resource use.
- Specifically, a small effect calculus based on the λ-calculus, with operations on resources.
- Rich enough to capture examples written in a subset of a capability-safe language, Wyvern.
- Look at how we can minimise the need for effect annotations in order to make such a system easier to use.
- Using capabilities allows us to bound the effects of unannotated code without needing to annotate it.

Basic Language

We start with a very simple language with operations on resources.

$$
\begin{array}{lcll}
e & ::= & & \textit{exprs}: \\
 & | & x & \textit{variable} \\
 & | & v & \textit{value} \\
 & | & e\ e & \textit{application} \\
 & | & e.\pi & \textit{operation} \\
 & & & \\
v & ::= & & \textit{values}: \\
 & | & r & \textit{resource literal} \\
 & | & \lambda x : \tau . e & \textit{abstraction}
\end{array}
\qquad
\begin{array}{lcll}
\tau & ::= & & \textit{types}: \\
 & | & \{\bar{r}\} & \textit{resource set} \\
 & | & \tau \rightarrow \tau & \textit{function} \\
 & & & \\
\Gamma & ::= & & \textit{type ctx}: \\
 & | & \varnothing & \textit{empty ctx} \\
 & | & \Gamma, x : \tau & \textit{binding}
\end{array}
$$

- Semantics uses reduction relation $e \longrightarrow e$ (ignoring operations).
- Type system has judgements: $\Gamma \vdash e : \tau$.
- Shows types of inputs and outputs, but nothing about effects.

Adding Effects

Add annotations to function types to show the effects that *may* occur.

$$
\begin{array}{lcll}
e & ::= & & \textit{exprs}: \\
 & | & x & \textit{variable} \\
 & | & v & \textit{value} \\
 & | & e\ e & \textit{application} \\
 & | & e.\pi & \textit{operation} \\
 \\
v & ::= & & \textit{values}: \\
 & | & r & \textit{resource literal} \\
 & | & \lambda x : \tau . e & \textit{abstraction}
\end{array}
$$

$$
\begin{array}{lcll}
\tau & ::= & & \textit{types}: \\
 & | & \{\bar{r}\} & \textit{resource set} \\
 & | & \tau \rightarrow_{\varepsilon} \tau & \textit{function} \\
 \\
\Gamma & ::= & & \textit{type ctx}: \\
 & | & \varnothing & \textit{empty ctx.} \\
 & | & \Gamma, x : \tau & \textit{binding} \\
 \\
\varepsilon & ::= & & \textit{effects}: \\
 & | & \{\overline{r.\pi}\} & \textit{effect set}
\end{array}
$$

Effects are sets of resource-operation pairs.

Adding Effects

- Semantics uses reduction relation $e \longrightarrow e \mid \varepsilon$, where ε is the effects that occur during evaluation of *e*.
- Type/effect system has judgements: $\Gamma \vdash e : \tau \ \texttt{with} \ \varepsilon$.

  So we can check what effects may occur during evaluation of *e*.
- But this requires extensive annotation, which is tedious in practice.

  E.g. Java unchecked exceptions are often criticised and often misused.
- Also, we may want to import third-party code which is not annotated.

Adding Capabilities

- Key idea is to combine annotated and unannotated code.
  - Allow annotated code to import unannotated code.
  - passing it the capabilities (resources) it needs.
  - and specifying the effects they are permitted to have.

- Combine the languages of unannotated and annotated code. using hat (e.g. $\hat{e}$) in the formalism to distinguish them.

- Add a new statement: $\texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e$
  - e is the unannotated code being imported
  - $\hat{e}$ is the capability being passed to e, which is bound to x in e.
  - ε_s is the set of effects which e is allowed to have ("selected authority").

- E.g. $\texttt{import}(\texttt{File.append})\ x = \texttt{File}\ \texttt{in}\ \lambda y : \texttt{Unit}.\ \texttt{x.write}$.

  This logger exceeds its authority so will be rejected!

Adding Capabilities

- Semantics uses reduction relation $\hat{e} \longrightarrow \hat{e} \mid \varepsilon$.

  We are only concerned with executing annotated code.

- To execute unannotated code which is imported, we annotate it with the selected authority.

$$\frac{}{\mathtt{import}(\varepsilon_s)\ x = \hat{v}\ \mathtt{in}\ e \longrightarrow [\hat{v}/x]\mathtt{annot}(e, \varepsilon_s) \mid \varnothing} \quad \text{(E-IMPORT2)}$$

  - $\mathtt{annot}(e, \varepsilon_s)$ just adds ε_s to function arrows in e.

Example: Importing Logger

```
let MakeLogger =
   (λf: File.
      import(File.append) f = f in
         λx: Unit. f.append) in

let MakeClient =
   (λlogger: Logger.
      λx: Unit. logger unit) in

let MakeMain =
   (λf: File.
      let loggerModule = MakeLogger f in
      let clientModule = MakeClient loggerModule in
      clientModule unit) in

MakeMain File
```

Note: `let` expression is usual syntactic sugar.

Example: Higher Order Effects

```
let malicious =
    (import(∅) y=unit in
        λf: Unit → Unit. f()) in

let plugin =
    (λf: {File}.
        malicious(λx:Unit. f.read)) in

let MakeMain =
    (λf: {File}.
        plugin f) in

MakeMain File
```

Type and Effect Checking for Imports

Most type and effect rules are straightforward, but…

For import, we want a rule of the form:

$$\frac{\cdots}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\; x = \hat{e}\; \texttt{in}\; e : \cdots\; \texttt{with}\; \cdots} \; (\varepsilon\text{-}\textsc{Import})$$

- What type and effects does the import expression have?
- What assumptions do we need?

Typing Imports – First Attempt

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT1})$$

- Assume arbitrary type and effect for $\hat{e}$.
- Must be able to type e, given just that x has type $\hat{\tau}$,
  to ensure e uses only the capabilities provided to it.
- e is unannotated while $\hat{\tau}$ is annotated, so we erase the annotations from $\hat{\tau}$.
- e has type τ — but τ is unannotated, so we annotate with ε_S.
- Evaluating e has all effects in ε_1 and ε_S.

Typing Imports – Second Attempt

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e} \texttt{ in } e : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT2})$$

- First version allows any capability to be passed to *e*.

- Restrict $\hat{e}$ so that its effects are contained in ε_s.

- *effects* collects all the effects captured by its argument.

$$\texttt{effects}(\{\bar{r}\}) = \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$$
$$\texttt{effects}(\hat{\tau}_1 \rightarrow_\varepsilon \hat{\tau}_2) = \texttt{effects}(\hat{\tau}_1) \cup \varepsilon \cup \texttt{effects}(\hat{\tau}_2)$$

```
import({File.*})
  go = λx: Unit →∅ Unit. x unit
  f = File
in
  go (λy: Unit. f.write)
```

Typing Imports – Third Attempt

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s \qquad \boxed{\texttt{ho-safe}(\hat{\tau}, \varepsilon_s)} \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e} \texttt{ in } e : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT3})$$

$$\texttt{effects}(\{\bar{r}\}) = \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$$
$$\texttt{effects}(\hat{\tau}_1 \rightarrow_\varepsilon \hat{\tau}_2) = \texttt{ho-effects}(\hat{\tau}_1) \cup \varepsilon \cup \texttt{effects}(\hat{\tau}_2)$$

$$\texttt{ho-effects}(\{\bar{r}\}) = \varnothing$$
$$\texttt{ho-effects}(\hat{\tau}_1 \rightarrow_\varepsilon \hat{\tau}_2) = \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$$

- Need to distinguish “direct” effects from “higher-order” effects.
- And ensure safe use of resources: imported capabilities must be expecting the effects they are passed by unannotated code.

Typing Imports – Fourth (and Final) Attempt

$$\frac{\boxed{\texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s} \\ \hat{\Gamma} \vdash \hat{e} : \hat{\tau} \ \texttt{with} \ \varepsilon_1 \quad \texttt{ho-safe}(\hat{\tau}, \varepsilon_s) \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s) \ x = \hat{e} \ \texttt{in} \ e : \texttt{annot}(\tau, \varepsilon_s) \ \texttt{with} \ \varepsilon_s \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT})$$

Distinction between direct and higher-order effects needs to be pushed further!

$\boxed{\mathbf{safe}(\hat{\tau}, \varepsilon)}$

$$\frac{\{r.\pi \mid r \in \bar{r}, \pi \in \Pi\} \subseteq \varepsilon}{} \ (\text{SAFE-RESOURCE})$$

$$\frac{\varepsilon \subseteq \varepsilon' \quad \mathbf{ho\text{-}safe}(\hat{\tau}_1, \varepsilon) \quad \mathbf{safe}(\hat{\tau}_2, \varepsilon)}{\mathbf{safe}(\hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2, \varepsilon)} \ (\text{SAFE-ARROW})$$

$\boxed{\mathbf{ho\text{-}safe}(\hat{\tau}, \varepsilon)}$

$$\frac{}{\mathbf{ho\text{-}safe}(\{\bar{r}\}, \varepsilon)} \ (\text{HOSAFE-RESOURCE})$$

$$\frac{\mathbf{safe}(\hat{\tau}_1, \varepsilon) \quad \mathbf{ho\text{-}safe}(\hat{\tau}_2, \varepsilon)}{\mathbf{ho\text{-}safe}(\hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2, \varepsilon)} \ (\text{HOSAFE-ARROW})$$

Conclusions

- We can now check examples like the ones given earlier and safely reject ones that violate the granted authority.
- Doesn't require programmers to add effect annotations.
- Relies on type checking, not effect checking — doesn't require unannotated expressions to be analysed for their effects.

Implementation by Justin Lubin as Undergraduate RA at CMU in the summer of 2018

APPROXIMATING POLYMORPHIC EFFECTS WITH CAPABILITIES

Goal

Allow *secure* and *ergonomic* mixing of effect-unannotated code with effect-annotated code in a *realistic* capability-safe programming language.

Object Capabilities

Capabilities

Unforgeable objects that give particular parts of the code access to sensitive resources

Capability-safe language

A language in which the only way to access sensitive resources is via capabilities

```
module def logger(myFile : File)
  ...

module def main(platform : Platform)
  val myFile = file(platform)
  val myLogger = logger(myFile)
  ...
```

Effect Systems

Effect system

Annotations on methods describing effects they can incur

Capability-based effect system

Way of formally reasoning about capabilities *(awesome!)*

Downside: verbosity

Capability-Safe Import Semantics

Prior work *(Craig et al.)*

Import semantics for capability-safe lambda calculus

Limitation

Does not handle mutable state nor effect polymorphism

Our goal

Scale up to a more realistic programming language

The Problem

Effect polymorphism *and* mutability

The Problem

```
resource type Logger
  effect log
  def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = ...
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

Question: *How will annotated code use **reversePlugin**?*

Effect polymorphism + mutability
⇒ **log** effect could be *anything!*

The Problem

```
resource type Logger
  effect log
  def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = ...
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

Question: *How will annotated code use **reversePlugin**?*

Effect polymorphism + mutability
⇒ **log** effect could be *anything!*

The Problem

```
resource type Logger
  effect log
  def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = ...
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

Question: *How will annotated code use **reversePlugin**?*

Effect polymorphism + mutability
⇒ **log** effect could be *anything!*

Solution

Quantification lifting

Quantification Lifting: Idea

```
resource type Logger
  effect log
  def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = ...
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

```
resource type Logger[effect E]
  def append(contents : String) : {E} Unit


module def reversePlugin[effect E](name : String)
  var logger : Logger[E] = ...
  def setLogger(newLogger : Logger[E]) : {E} Unit
    logger = newLogger
  def run(s : String) : {E} String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

- ***Lift*** effect polymorphism from inside ML-style module functor to the functor itself
- Collapse each universal effect quantification into single quantified effect E
  - Serves as effect bound for all methods in module

Quantification Lifting: Idea

```
resource type Logger
  effect log
  def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = ...
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

```
resource type Logger[effect E]
  def append(contents : String) : {E} Unit

module def reversePlugin[effect E](name : String)
  var logger : Logger[E] = ...
  def setLogger(newLogger : Logger[E]) : {E} Unit
    logger = newLogger
  def run(s : String) : {E} String
    val t = s.reverse()
    logger.append(name + ": " + s + " -> " + t)
    t
```

- ***Lift*** effect polymorphism from inside ML-style module functor to the functor itself
- Collapse each universal effect quantification into single quantified effect E
  - Serves as effect bound for all methods in module

Quantification Lifting: Usage

```
import fileLogger, databaseLogger, reversePlugin
val logger1 = fileLogger(...)
val logger2 = databaseLogger(...)
val plugin = reversePlugin[logger1.log](“archive”)
def main() : {logger1.log} Unit
  plugin.setLogger(logger1)
  // plugin.setLogger(logger2) <-- not allowed!
```

```
resource type MyPlugin
  def setLogger(newLogger : Logger’) : {logger1.log} Unit
  def run(s : String) : {logger1.log} String

resource type Logger’
  effect log = {logger1.log}
  def append(contents : String) : {log} Unit
```

Quantification Lifting: Import Bounds

```
resource type Logger
  effect log
  def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = ...
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    logger.append(name + “: ” + s + “ -> ” + t)
    t
```

```
resource type Logger[effect E]
  def append(contents : String) : {E} Unit


module def reversePlugin[effect E](name : String)
  var logger : Logger[E] = ...
  def setLogger(newLogger : Logger[E]) : {E} Unit
    logger = newLogger
  def run(s : String) : {E} String
    val t = s.reverse()
    logger.append(name + “: ” + s + “ -> ” + t)
    t
```

- ***Something to be careful about:*** bounds on new universally-quantified polymorphism
  - *Upper bound:* Craig et al. import semantics
  - *Lower bound:* Capability-safety

Quantification Lifting: Type-Level Transformation

Benefit

Don't need code ahead of time, only type signature

- Dynamic loading (plugins)
- Compiled code
- Third-party libraries

Drawback

Over-approximation of possibly-incurred effects

Quantification Lifting: Type-Level Transformation

Before: $\tau_1 \rightarrow \tau_2$

After: $\forall \varepsilon \; (L \subseteq \varepsilon \subseteq U) \; . \; \tau_1 \rightarrow (\tau_2)_\varepsilon$

Related Work

Effect inference

- Operates on *expressions*
- Gives exact bound on effects that can be incurred

Algebraic effects

- Has a different goal
- We use the effect system to formally/statically reason about capabilities

Observations

- ***Capabilities*** are good way of managing non-transitive access to system resources
- ***Effect systems*** can formalize capability-based reasoning, but can be verbose
- Craig et al.'s ***import semantics*** work great for lambda calculus
- ***Quantification lifting*** handles tricky interaction between effect polymorphism and mutable state

Example Summary

```
resource type Logger
 effect log
 def append(contents : String) : {log} Unit

module def reversePlugin(name : String)
 var logger : Logger = ...
 def setLogger(newLogger : Logger) : Unit
  logger = newLogger
 def run(s : String) : String
  val t = s.reverse()
  logger.append(name + ": " + s + " -> " + t)
  t
```

```
resource type Logger[effect E]
 def append(contents : String) : {E} Unit

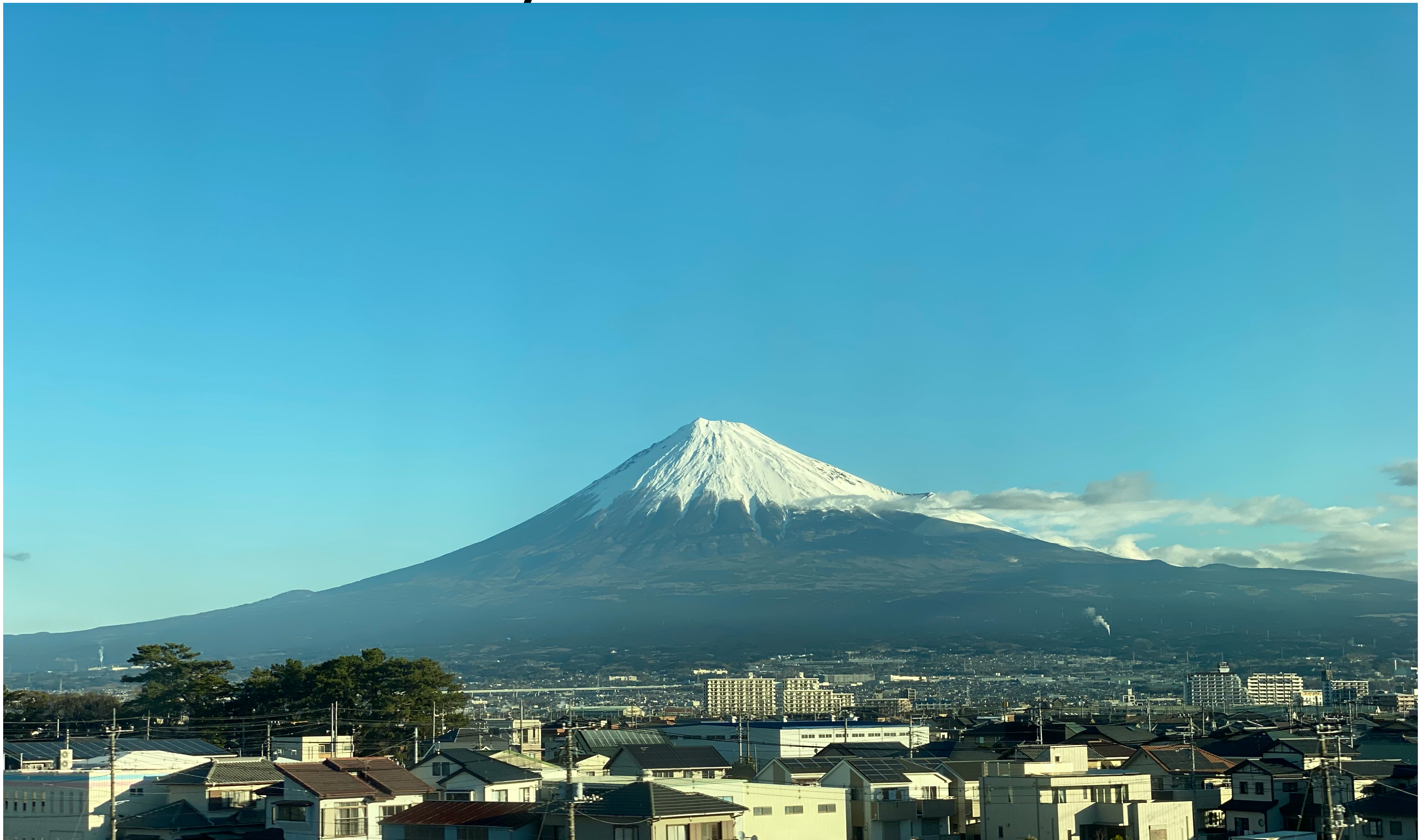

module def reversePlugin[effect E](name : String)
 var logger : Logger[E] = ...
 def setLogger(newLogger : Logger[E]) : {E} Unit
  logger = newLogger
 def run(s : String) : {E} String
  val t = s.reverse()
  logger.append(name + ": " + s + " -> " + t)
  t
```

```
import fileLogger, databaseLogger, reversePlugin
val logger1 = fileLogger(...)
val logger2 = databaseLogger(...)
val plugin = reversePlugin[logger1.log]("archive")
def main() : {logger1.log} Unit
 plugin.setLogger(logger1)
 // plugin.setLogger(logger2) <-- not allowed!
```

```
resource type MyPlugin
 def setLogger(newLogger : Logger') : {logger1.log} Unit
 def run(s : String) : {logger1.log} String

resource type Logger'
 effect log = {logger1.log}
 def append(contents : String) : {log} Unit
```

Thank you for the course!



CAPITAL THINKING.
GLOBALLY MINDED.
MAI I TE IHO KI TE PAE

VICTORIA UNIVERSITY OF
WELLINGTON
TE HERENGA WAKA
NEW ZEALAND