# Object Capabilities, Effects, and Abstraction

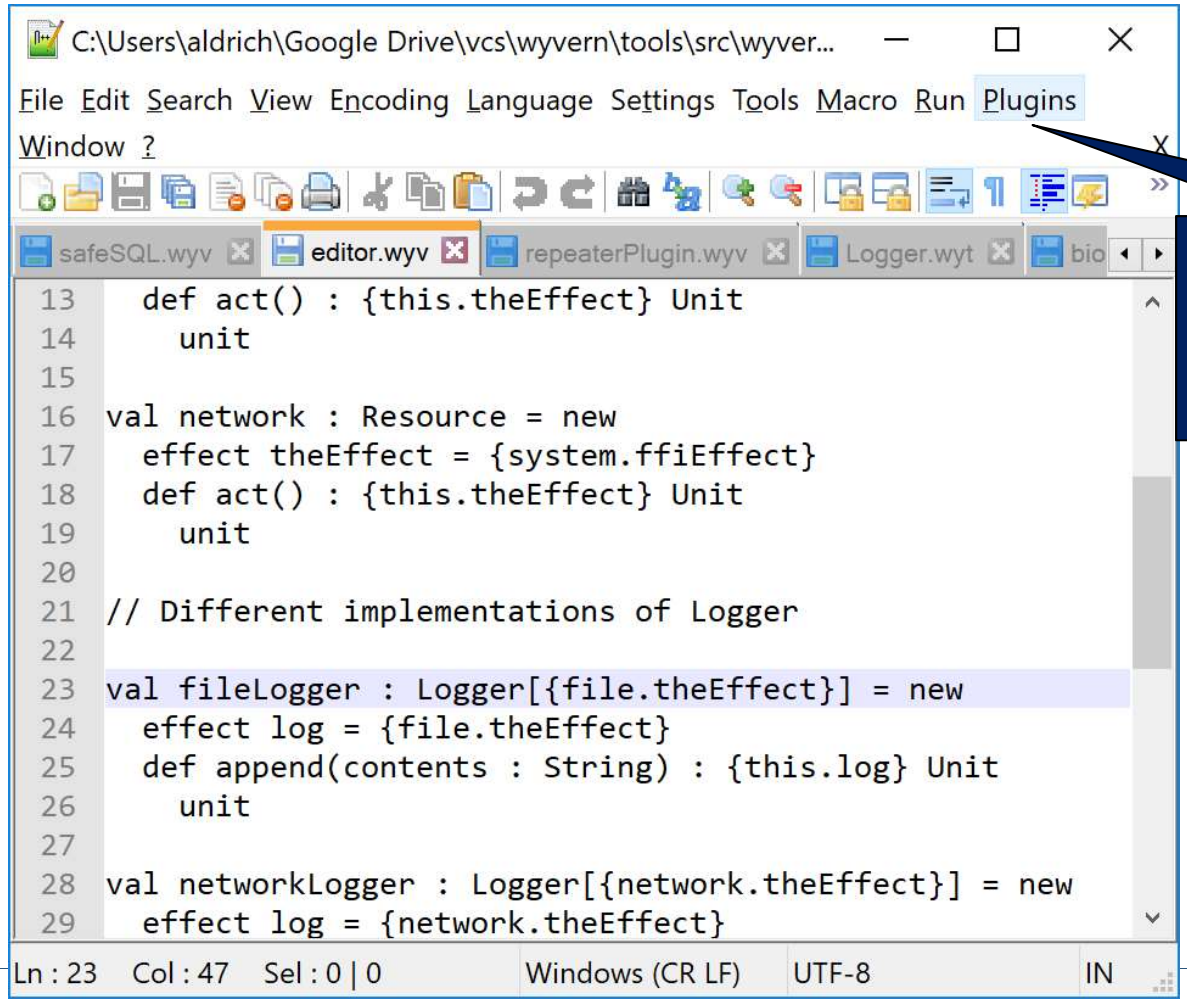
**Jonathan Aldrich**
aldrich@cs.cmu.edu
http://www.cs.cmu.edu/~aldrich/

**contributions from:**
Aaron Craig
Justin Lubin
Darya Melicher
Alex Potanin
Anlun Xu
Valerie Zhao

INRIA Prosecco Seminar
November 26, 2018


# What might code do to system resources?

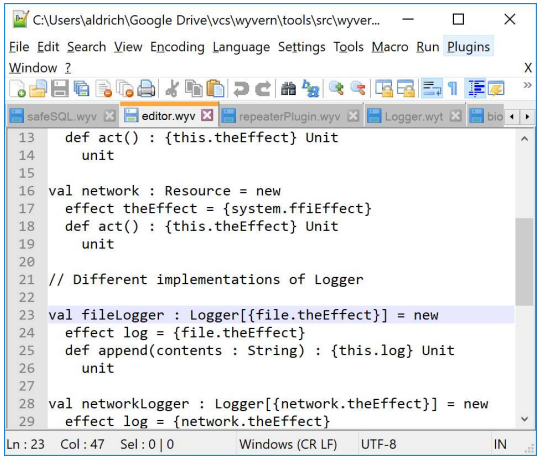
- Goal: constrain plugins




Are these plugins safe? Could a malicious plugin delete my files?

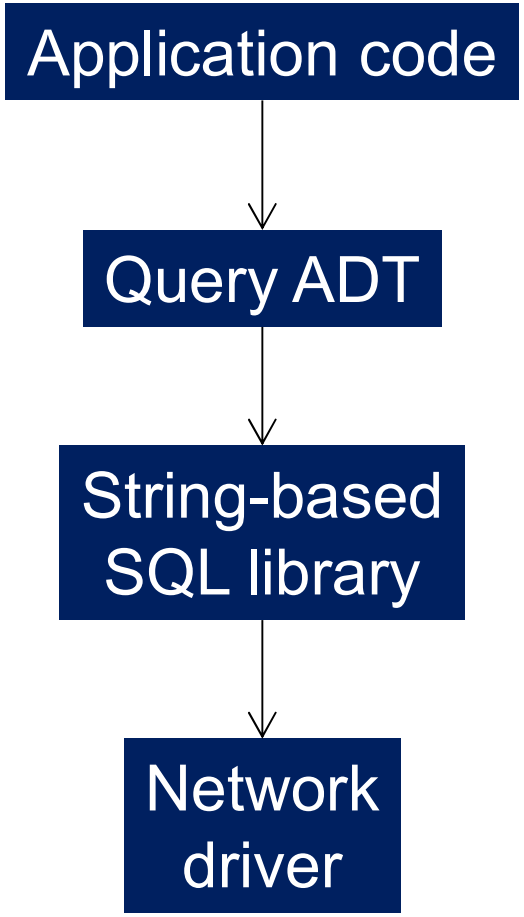
# What might code do to system resources?

- What can a plugin do?
  - File reads, writes?
  - Log an error message?
    - Like a file write, but more ***abstract***

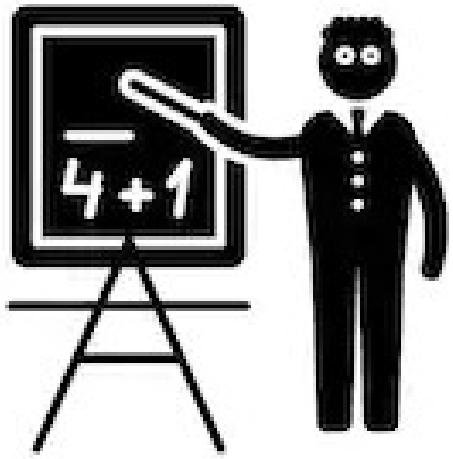



# What might code do to system resources?

- Goal: enforce architectural layering [Dijkstra 1968]
  - **Application code** gets data using a
  - **Query ADT**, which is implemented via a
  - **SQL library**, which sends commands via a
  - **Network driver**, which talks to database
- Security constraints
  - Network used only for database queries
  - No direct use of string-based SQL library
    - Possible command injection
- **Abstraction** of system resources
  - Application does ***safe queries***
  - Query ADT does ***unsafe queries***
  - Network driver does ***network access***



# This Talk

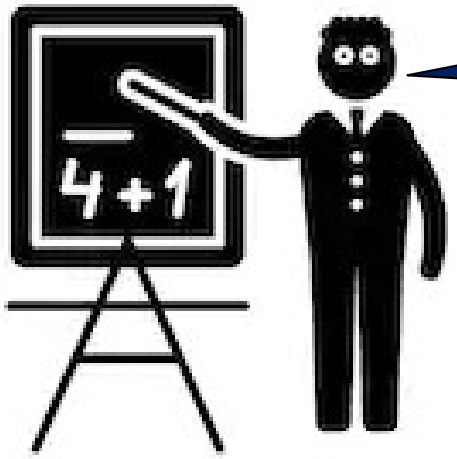



How to reason formally about resource use?

How to practically enforce resource use?

Can we be just as **lightweight** as the practical approach, but **retain formal reasoning**?

# Formalist: Let's check this statically!




An **effect system** sounds like a good idea!

# Checking Resource Use with Effects

- Effect systems [Lucassen & Gifford 88]
  - Annotate each function with its effects
  - Our setting: effects are **actions** on **resources**
    - `{passwordFile.read, wikipediaURL.request, …}`

Note: not algebraic effects

```
type File
    effect read
    def readContents(): {this.read} String

module def myPlugin(f:File)

def countWords() : {f.read} Int
    val contents = f.readContents()
    contents.split(' ').size()
```

`File` resources define a `read` effect (cf. type members in Scala)

`readContents` has a `read` effect on `this` file

`myPlugin` is a functor that is instantiated with a `File`

`readContents` has effect `f.read`, so `countWords` does too

# Checking Architectural Layering

```
// in signature of the rawSQL module
effect UnsafeQuery
type Connection
def connect(…) : Connection
def query(q:String) : {UnsafeQuery} SQLResult
```

The unsafe SQL library defines an UnsafeSQL effect

Query operations have an UnsafeQuery effect

```
// client code
def getData(input : String) : Data
  rawSQL.query("SELECT * FROM Students WHERE name ='“
                  + input + "';“)
```

Error: getData() must declare effect rawSQL.UnsafeQuery

Has effect rawSQL.UnsafeQuery

# Checking Architectural Layering

```
// in signature of the rawSQL module
effect UnsafeQuery
type Connection
def connect(…) : Connection
def query(q:String) : {UnsafeQuery} SQLResult
```

The unsafe SQL library defines an UnsafeSQL effect

Query operations have an UnsafeQuery effect

All dangerous code marked with effect

```
// client code
def getData(input : String) : {rawSQL.UnsafeQuery} Data
  rawSQL.query("SELECT * FROM Students WHERE name ='“
                + input + "';“)
```

Has effect rawSQL.UnsafeQuery

# Effect Abstraction

- Issue: won't users of the safeSQL library have an UnsafeQuery effect, if safeSQL is built on rawSQL?

The safeSQL functor uses a rawSQL module

```
module def safeSQL(rawSQL:RawSQL) : SafeSQL
type SQL
    val command : String
    …
effect SafeQuery = rawSQL.UnsafeQuery
def query(SQL) : {SafeQuery} SQLResult
    …
```

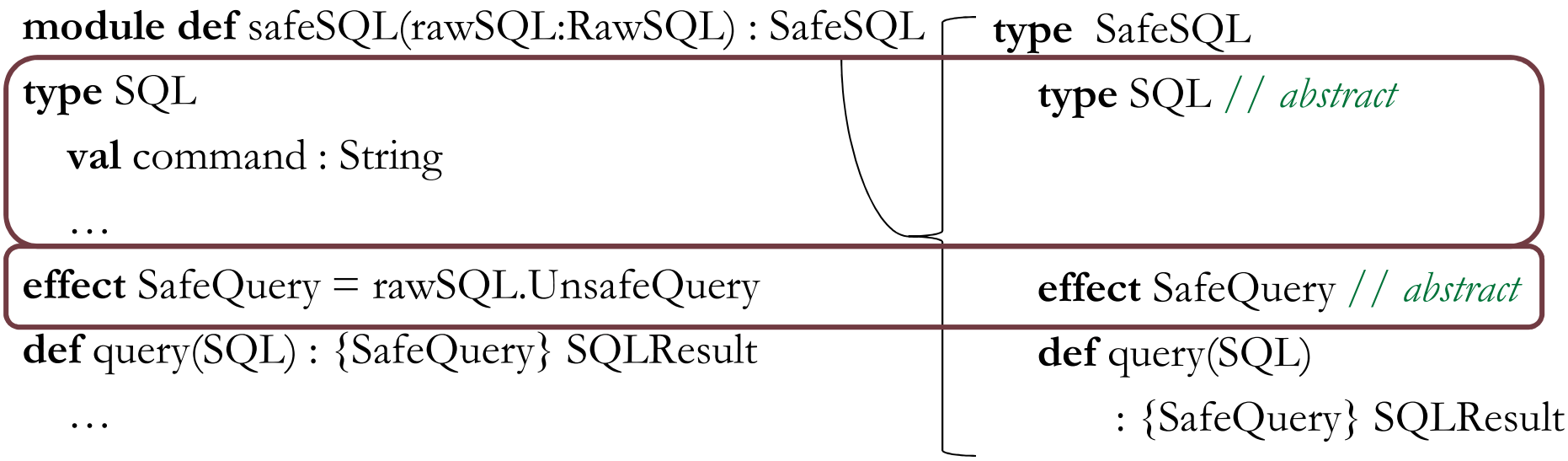
Defines a SQL ADT

The SafeQuery effect is defined in terms of UnsafeQuery.

Now clients have effect safeSQL.SafeQuery

# Opaque Signature Ascription

- We can hide the definition of an effect
  - Abstract effects – analogous to abstract types in ML modules [MacQueen, 1986]



# Effect abstraction → Effect polymorphism

- We can now encode effect polymorphism
  - Cf. type members → type polymorphism in Scala (and Wyvern)

```
// with polymorphism sugar
def twice[effect E](f : Int -> {E} Int, x:Int) : {E} Int
    f(f(x))
twice[file.write](…)
// the desugared version, using only effect members
type EffectHolder
    effect E
def twice(e:EffectHolder, f : Int -> {e.E} Int, x:Int) : {e.E} Int
    f(f(x))
twice(new EffectHolder{effect e=file.write}, …)
```

# Abstraction all the way down

- Primordial effect: the foreign function interface
  - All other I/O effects defined in terms of this
  - Can also "fake" an effect (no underlying I/O)

```
module net(java:Java)
import java:wyvern.utils.net.javaPacketUtils

effect network = java.ffiEffect
type Packet
  …
def send(p:Packet) : {network} Unit
  javaPacketUtils.send(p)
```

The net functor takes the Java FFI object as an argument

Methods imported from Java automatically have the java.ffiEffect effect

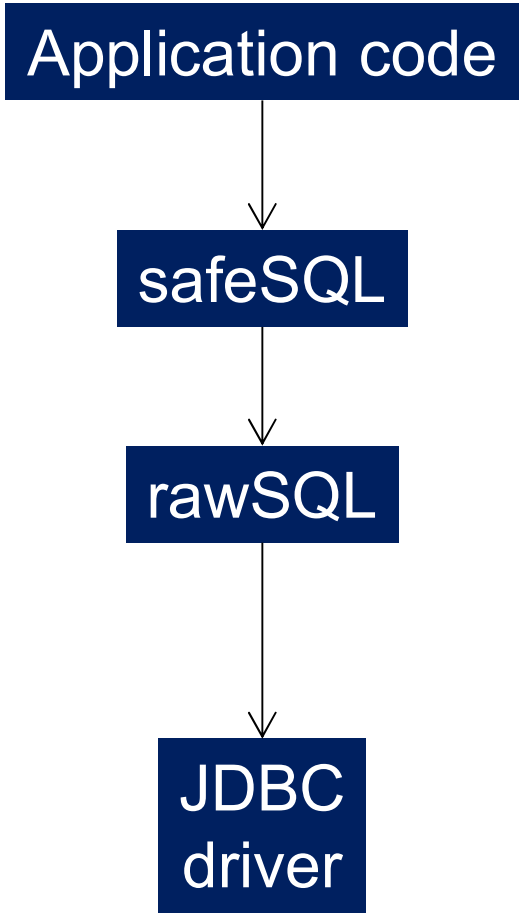
network is an abstract effect defined in terms of java.ffiEffect

Calls a Java function, therefore has ffiEffect; abstracted to clients as a network effect

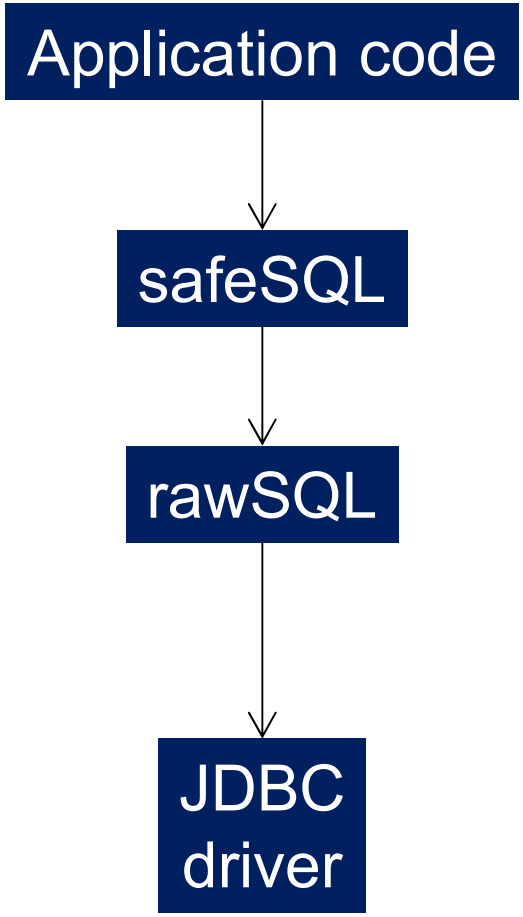
# Is Abstraction Safe?

- The safeSQL abstracts the UnsafeQuery effect as a SafeQuery effect
  - OK; safeSQL protects from command injection
  - But couldn't a malicious library also do this?

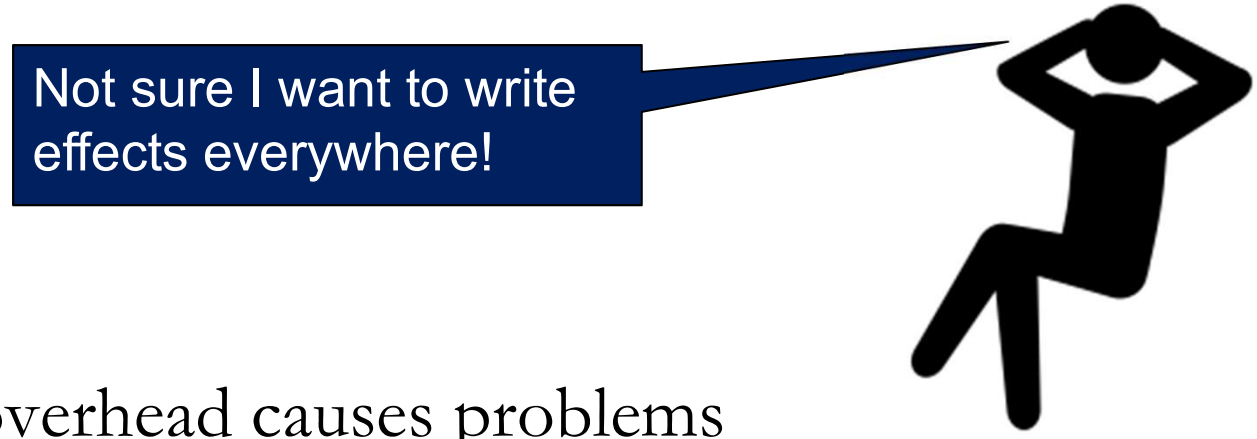
- Idea: analyze code for abstraction
  - UnsafeQuery is abstracted only by safeSQL, and only as a SafeQuery
  - Currently formalizing and implementing this



# Demo – Effects in Wyvern



# Two much annotation!



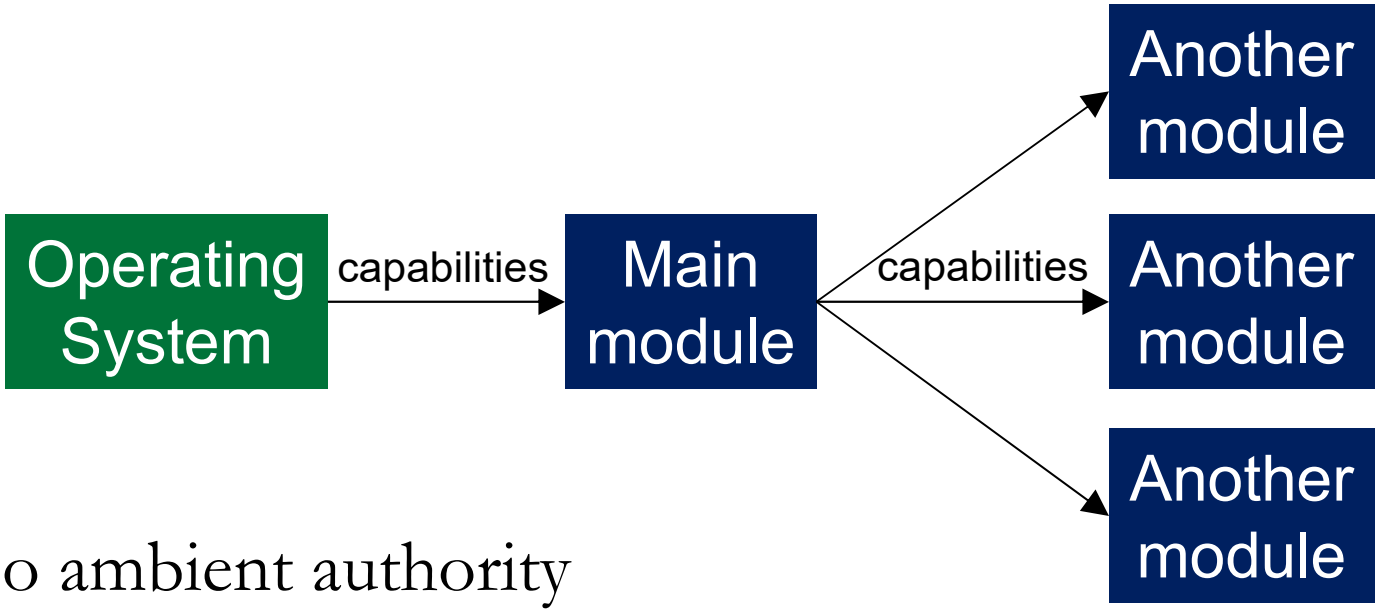

- Effect system overhead causes problems
  - E.g. Java's checked exceptions
  - Routinely bypassed using unchecked exceptions
- Inference [Talpin&Jouvelot 1994] is a solution, but has tradeoffs
  - Avoids need to annotate in many places
  - Only applies if you have the code
  - Usability issues
    - can fail because of something deep in the code
    - can succeed, then fail if the code changes

# Object capabilities

- Capability: an unforgeable token controlling access to a resource [Dennis & Van Horn 1966]
- Benefits
  - Simple approach to security
  - Principle of least privilege [Saltzer 1974]
- Object capability: object reference grants access to resource
  - Recent research: E [Miller 2006], Joe-E [Mettler et al. 2010]
  - Frozen Realms proposal for JavaScript

- Note: our capabilities are object references (dynamic)
  - *Capability* has also been used for permissions in a type system (static)

# Capability Safety




- No ambient authority
  - Code cannot use system resources by default
  - Main gets a capability to system resources from OS
  - Other modules can use resources only if main passes it to them
    - Pass only the capabilities each module needs
    - Often at functor instantiation time
  - Formalized as *capability safety* [Maffeis et al. 2010]
    - Adapted for Wyvern [Melicher et al. 2017]

# Capability Safety

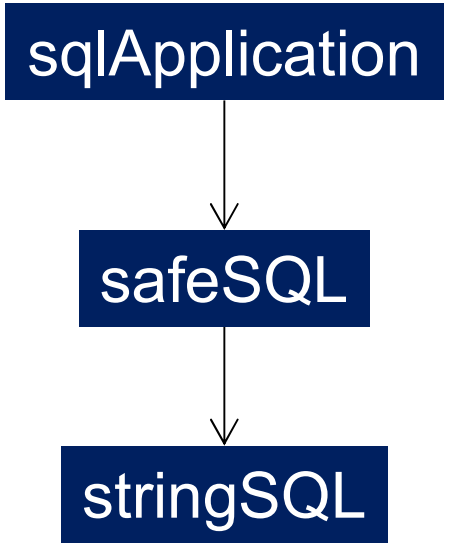
```
// the safeSQL module
module def safeSQL(stringSQL : db.StringSQL)
type SSQL = …
type Sselect = …
def select(row:String):SQLSelect =
…
// the main program
require db.stringSQL

import db.safeSQL
import modules.sqlApplication

val sql = safeSQL(stringSQL)
val app = sqlApplication(sql)

app.run()
```

safeSQL can't just import StringSQL; it must be passed a capability

main requires a stringSQL capability from the OS

These modules are pure (stateless) functors and can be imported

We instantiate the modules, passing capabilities



# Capabilities all the way down

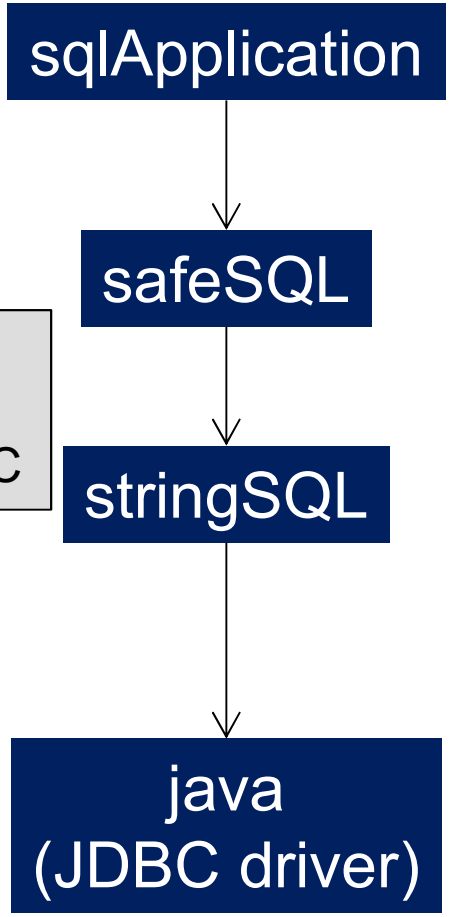
- The primordial capability is the FFI
  - "**require** db.stringSQL" instantiates stringSQL with a java FFI capability:

```
// the stringSQL module
module def stringSQL(java : Java)
import java:wyvern.JDBC.utils
type SQL
    def query():String
…
// the main program, now desugared
require java

import db.stringSQL
val ssql = stringSQL(java)
…
```

stringSQL is a functor that needs a java FFI capability to use JDBC

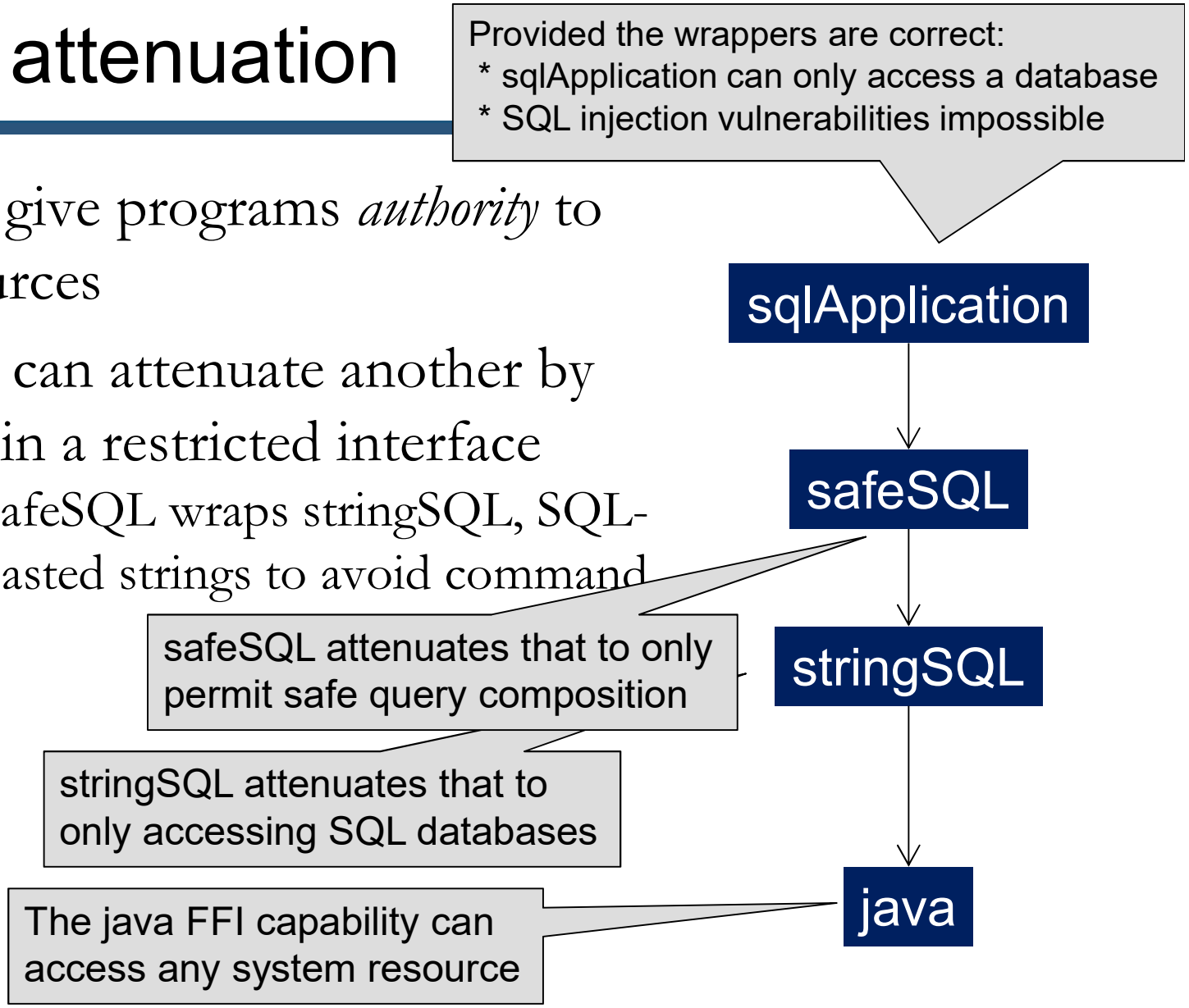
main requires a java FFI capability

Import and instantiate stringSQL, then the other modules

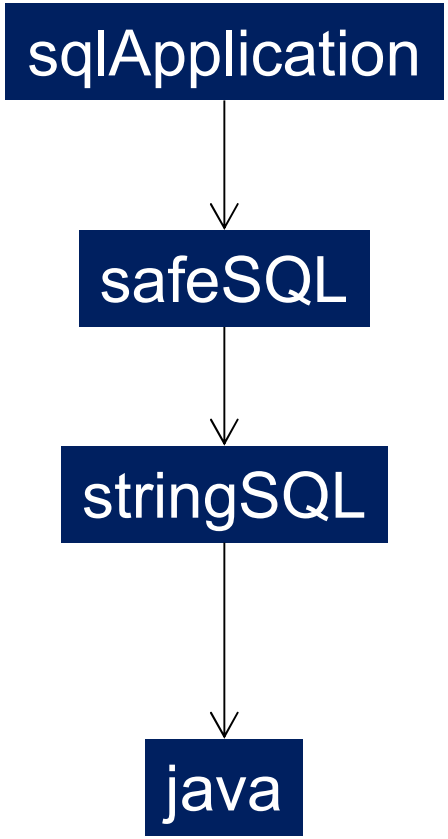


# Authority attenuation

- Capabilities give programs *authority* to access resources
- A capability can attenuate another by wrapping it in a restricted interface
  - Example: safeSQL wraps stringSQL, SQL-encoding pasted strings to avoid command injection



# Demo – Capabilities in Wyvern



# Comparing these mechanisms

**Effects**

- Static, mechanical reasoning
  - Effect annotation on a function
- Requires annotations
  - Can be burdensome
- Support abstraction
  - SafeQuery abstracts UnsafeQuery
- Build up from java.ffiEffect

**Capabilities**

- Reason using dynamic semantics
  - Who has which capability?
- No annotations required
  - But must pass capabilities explicitly
- Support attenuation
  - safeSQL attenuates stringSQL
- Build up from java

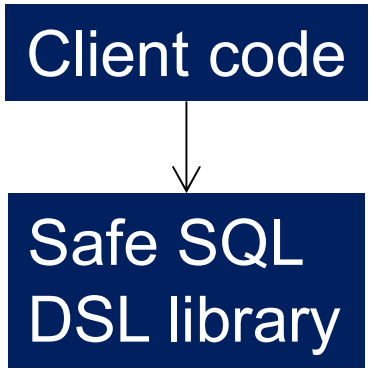
Capabilities are neat…but what do they really buy us?
Formally, no one really knows.

Idea: can we **combine** these mechanisms?

- Nearly as **lightweight** as capabilities
- Retain **formal reasoning** of effects
- Explain, formally, **why capabilities are useful**!

# Bounding effects with capabilities

- We can **bound a module's effects** by its **capabilities**
  - No need to effect-annotate the module
  - Note: requires capability safety!



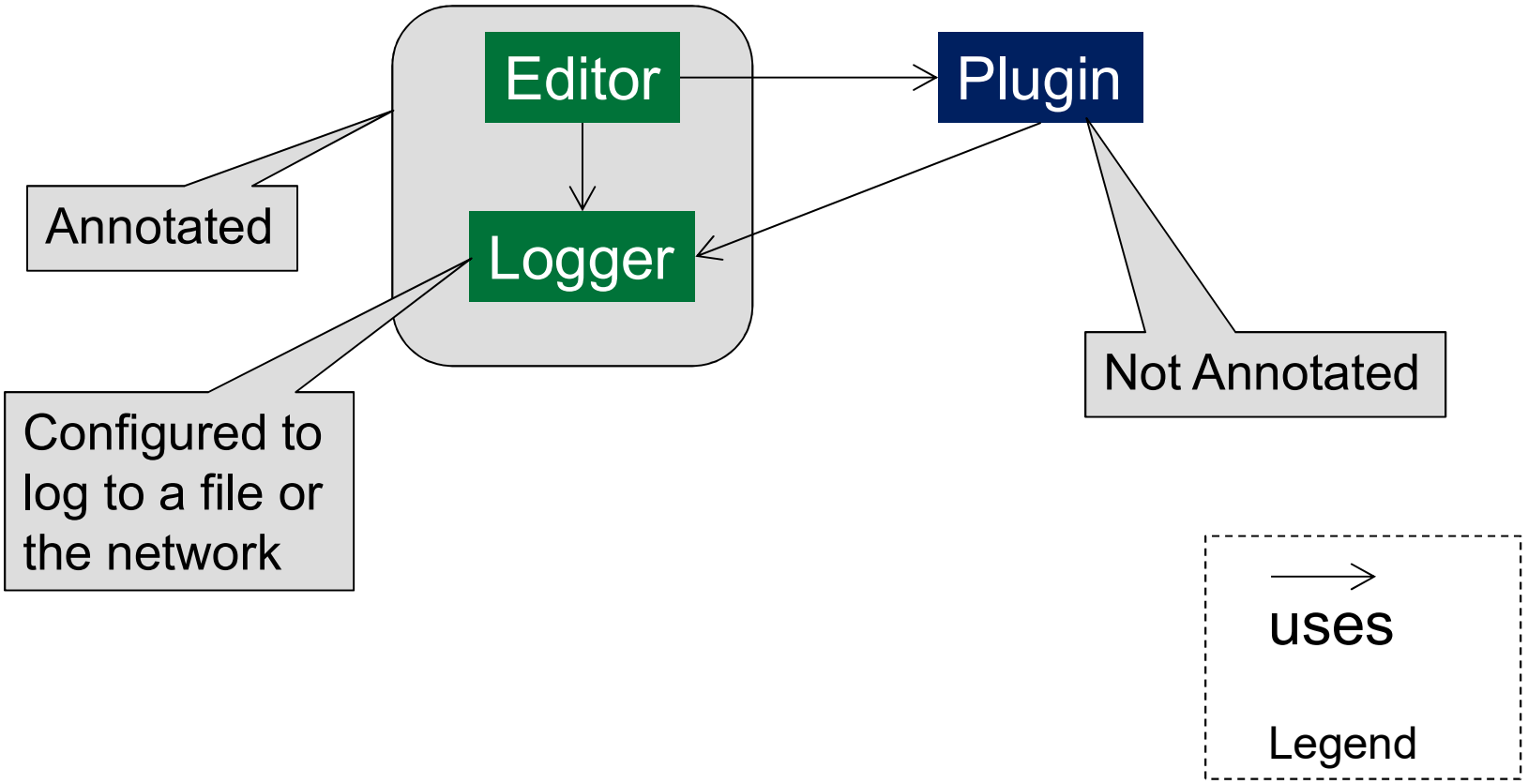

Client can have effect safeSQL.SafeQuery (and nothing else)

```
module def client(safeSQL : SafeSQL) : Client
import …

// other code
```

Imports must be pure - no effects

Scales up – most application code may be unannotated

# Effect bounding demo



# A Touch of Formality

- Lambda calculus – with and without effect annotations

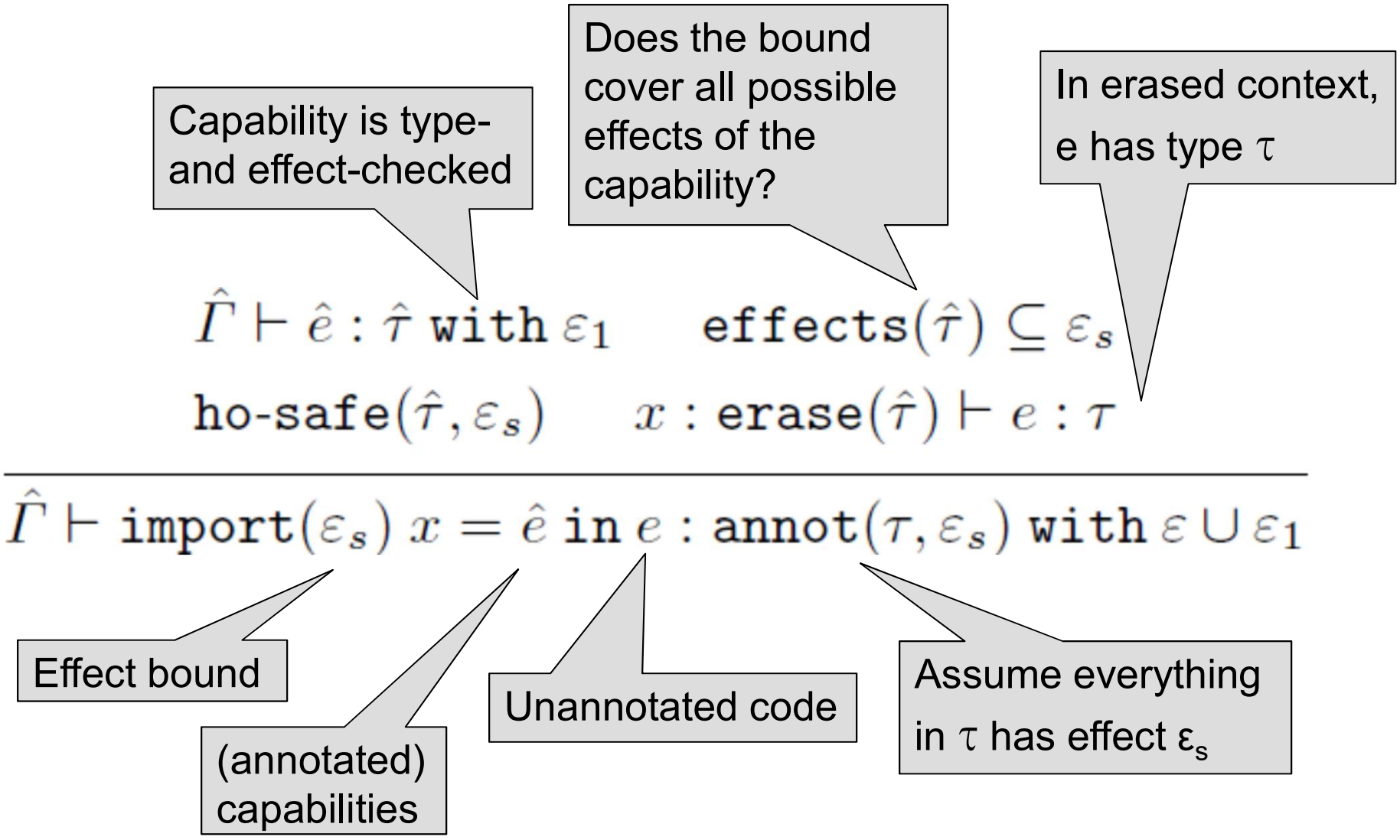
$$
\begin{array}{lll}
e ::= & & \textit{exprs}: \\
\mid & x & \textit{variable} \\
\mid & v & \textit{value} \\
\mid & e\ e & \textit{application} \\
\mid & e.\pi & \textit{operation}
\end{array}
\qquad
\begin{array}{lll}
\tau ::= & & \textit{types}: \\
\mid & \{\bar{r}\} & \textit{resource set} \\
\mid & \tau \to \tau & \textit{function}
\end{array}
$$

Unannotated types

$$
\begin{array}{lll}
\hat{e} ::= & & \textit{labelled exprs}: \\
\mid & x & \textit{variable} \\
\mid & \hat{v} & \textit{value} \\
\mid & \hat{e}\ \hat{e} & \textit{application} \\
\mid & \hat{e}.\pi & \textit{operation} \\
\mid & \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e & \textit{import}
\end{array}
\qquad
\begin{array}{lll}
\hat{\tau} ::= & & \textit{annotated types}: \\
\mid & \{\bar{r}\} & \textit{resource set} \\
\mid & \hat{\tau} \to_{\varepsilon} \hat{\tau} & \textit{function}
\end{array}
$$

Effect annotation

# Including unannotated code in annotated context




Full formalization and effect system soundness in ICFEM 2018

# Effect computation: examples

1. effects(**Unit** $\rightarrow_{\mathbf{A}}$ **Unit**) = {A}
2. effects(**Unit** $\rightarrow_{\mathbf{A}}$ **Unit** $\rightarrow_{\mathbf{B}}$ **Unit**) = {A,B}
   - Client can have effect A and then effect B
3. effects((**Unit** $\rightarrow_{\mathbf{A}}$ **Unit**) $\rightarrow_{\mathbf{B}}$ **Unit**) = {B}
   - Client can have effect B, but does not gain the ability to have effect A because the client produces the argument
4. effects(((**Unit** $\rightarrow_{\mathbf{A}}$ **Unit**) $\rightarrow_{\mathbf{B}}$ **Unit**) $\rightarrow_{\mathbf{C}}$ **Unit**) = {A,C}
   - Client can have effect C by calling the function, and effect A by providing a function that gets a (Unit $\rightarrow_{\mathrm{A}}$ Unit) and invokes it

# Effect computation: examples **and rule**

1. effects(Unit $\rightarrow_A$ Unit) = {A}
2. effects(Unit $\rightarrow_A$ Unit $\rightarrow_B$ Unit) = {A,B}
3. effects((Unit $\rightarrow_A$ Unit) $\rightarrow_B$ Unit) = {B}
4. effects(((Unit $\rightarrow_A$ Unit) $\rightarrow_B$ Unit) $\rightarrow_C$ Unit) = {A,C}

Examples 3 & 4

Example 1

Example 2

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

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

Example 4

(similar to Example 2)

# Including unannotated code in annotated context

Make sure the unannotated code can't break annotated code

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

Note: complete formal system adds one more check that models primitive capabilities – see ICFEM '18 paper

# Why the higher-order safety check?

**import**(A) x : (Unit →$_{\{\}}$ Unit) →$_{A}$ Unit **in**

**let** f = x(id) **in** x(f) // *unannotated*

- Unsafe!
  - unannotated code can have effect A
  - therefore can pass a lambda with effect A to the imported function, which assumes its argument has no effect

No check on the lambda's annotation ε'

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

But we do check the annotation on the argument

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

# Lifting effect polymorphism

```
module def repeaterPlugin(defaultLogger : Logger)

  var logger : Logger = defaultLogger
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
```

Our solution is *lifts* poly-morphism to the module level, where the state is created

We would like to assign this function an effect-polymorphic type

But the function might assign to local state, so the effect of newLogger must be bounded by the overall effect of the module (*cf.* polymorphism and state, more generally)

# Comparison and tradeoffs vs. inference

**Effect Inference**

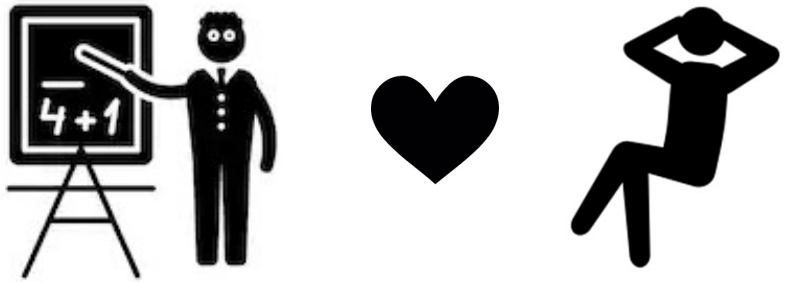
- Only requires effects
- Requires code
- Failure may depend on code
  - E.g. large components, evolution

**Effect Bounds**

- Requires capabilities & effects
- Requires only type
- Failure depends only on type
  - E.g. large components, evolution
- Explains value of capabilities for formal reasoning

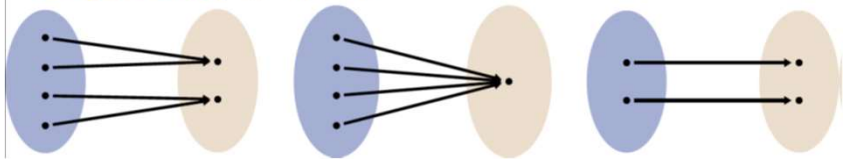
# Capabilities and Effects: So Happy Together

- Goal: controlling system resources
- Two approaches
  - Effect system, supporting abstraction
  - Capability safety, supporting attenuation
- A link between them: bounding effects with capabilities
  - Use lightweight approach for many components
  - Formally reason about the effect of all components
  - Bonus: formal explanation of the reasoning power of capabilities



# Ask me about…

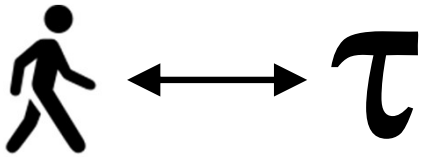
```
Definition Surjection(Map f, Set X, Set Y):
    forall y : Y | exists x : X | f(x) = y
f: A -> B
Set A, B
Surjection(f, A, B)
```



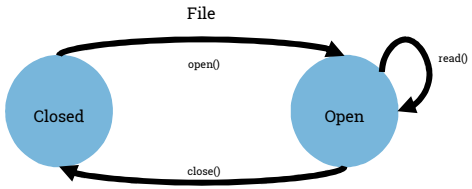
**Penrose**: write math, get diagrams

**requires** ? ∧ balance > 0

**Gradual verification** (with Éric)



**Glacier:** usable immutability types




**Obsidian:** usable typestate for smart contracts

**Hazel:** type theory for IDEs

More on **Wyvern**

- Run your architecture
- Language extensions as libraries