Achieving Architectural Control via Language Support for Capabilities

Jonathan Aldrich

aldrich@cs.cmu.edu

http://www.cs.cmu.edu/~aldrich/

contributions from:

Darya Kurilova
Joseph Lee
Troy Shaw
Esther Wang
Alex Potanin
Cyrus Omar
Yangqingwei Shi
Du Li

IFIP Working Group on Language Design

January 2016

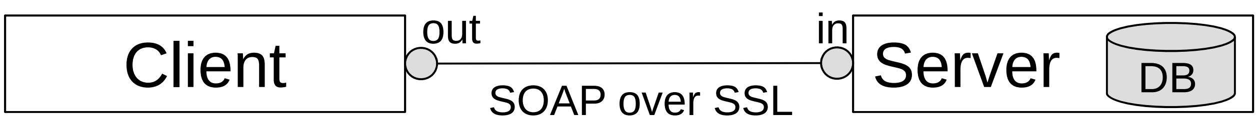
institute for SOFTWARE RESEARCH

Carnegie Mellon University
School of Computer Science

Do You Control Your Architecture?

- ***Architectural Control*** [AOPL14] is the ability of the software architect to:
  - **Specify** architectural constraints sufficient to ensure system properties
  - **Enforce** those constraints as the system is built and evolved

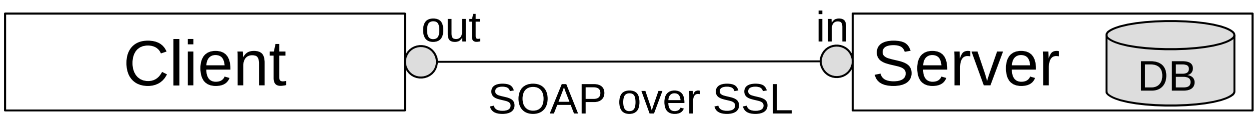
- Distributed system example:



- Is the architecture followed consistently in the system?
  - Does the implementation always use SSL?
  - Does the implementation add any hidden connections?
  - Does the client access the disk, or is it stateless (as shown)?

Architectural Control is Hard

- Distributed system example in Java



- In Java code, is SSL used consistently? Are there other connections?
  - Does dynamically loaded code use the network?
  - What about third party libraries, or native code?
- Many architectural properties similarly depend on use of resources
  - Network, storage, etc. – if the OS controls it, the architect may want to also
- Today constraints are enforced (imperfectly) via software process
  - Each developer must know and follow the architectural rules during evolution
  - Assuring third-party code is difficult
    - Sandboxing is one possible technique – but difficult and error-prone in practice [CMD+15]

A Vision: Own Your Architecture

1. Resource architecture
   - A specification of which modules can use key resources
     - Example: only the Middleware module can use the network
   - Resources include I/O and global state, but additional resources can be defined
   - Enforced by built-in language mechanisms

Capabilities [DV66]

2. Delegate enforcement of properties
   - Owners of modules that access the relevant resource
     - Example: the architect discusses important communication invariants with the middleware lead, and important storage invariants with the database lead

3. Keep architecture under version control
   - Architect approves all changes
     - Example: if a developer requests access to the network, the architect can approve—or more likely, tell the developer to use the existing middleware library

Capability-Based Resource Control

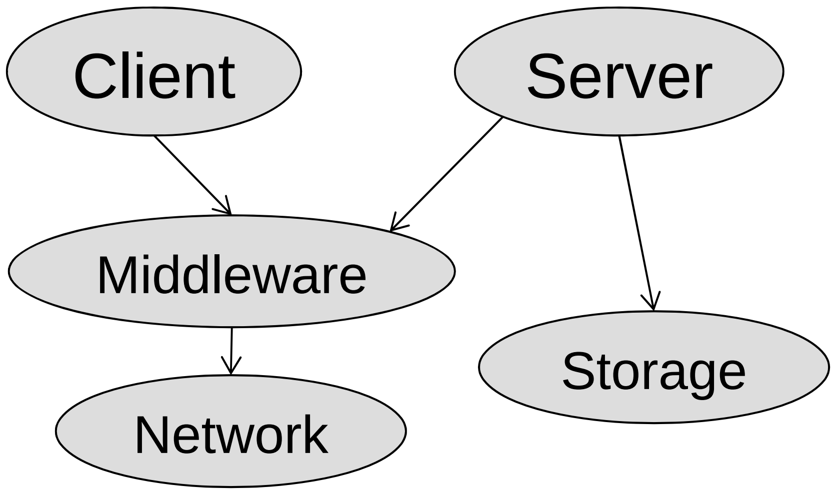
Is **owning** the architecture sufficient to **control it**?

- What if the Client opens other, unsecured, connections?

Conceptual Architecture [SG94]

Solution: resources as **capabilities**

- Capability: an unforgeable token controlling access to a resource [DV66]
- No ambient capabilities
  - By default, Client and Server have no network capability
- Capability delegation
  - Explicitly pass capabilities to modules, such as Middleware, that need them



Capability/Object Structure

Capability-Safe Languages [Mil06]

- A language in which an object can only get a capability if it is explicitly given one

- Restrictions
  - No ambient authority – resources unavailable without a capability
    - E.g. cannot simply import java.io.* and then open a file
  - No global mutable state – would allow anyone to get/put capabilities
    - Global variables are OK if they hold transitively immutable values
  - Prior work: E, Joe-E, …

Capability-Safe Languages

- A language in which an object can only get a capability if it is explicitly given one

- Our research
  - A way to achieve architectural control via capabilities (outlined above)
    - Future work: validation, extension to IDEs and other tools
  - A capability-safe **module system**
    - Reconciles conveniences of typical module systems with capability safety
  - A **formalization** of capabilities in the presence of mutable state
    - Clarifies the role of mutable state in capability safety
  - A refined, **non-transitive notion of authority**
    - Supports informal reasoning about capability restriction
    - Future work: formalize this reasoning
  - Design principles for **capability-safe type systems** and **reflection**
    - Prior work focused on dynamically typed languages (E) or adapting existing designs (Joe-E)

Capability-Safe Module System

```
module Lists
  type List
    ...
  def create():List
    ...

resource module UserInfo
  import Lists
  var name : String = 'EMPTY'
  def init(uName : String) : Unit
    name := uName
    var actionHistory : Lists.List = Lists.create()
    ...
  def getName() : String = name
    ...


resource module DocumentLock
  require SigUserInfo as uInfo
  def sign() : Boolean =
    var signee : String = uInfo.getName()
    ...
  ...

resource module Main
  instantiate UserInfo() as uInfo
  instantiate DocumentLock(uInfo) as docLock
  ... // client code
```

Resource modules capture state or I/O;
Pure modules don't

Pure modules can be imported freely;
Resource modules must be required parameters of the client module

A resource module can be instantiated, passing in parameters (if any)
(*cf.* ML, Units, Newspeak, etc.)

Capability-Safe Module System

```
module Lists
  type List
    ...
  def create():List
    ...

resource module UserInfo
  import Lists
  var name : String = 'EMPTY'
  def init(uName : String) : Unit
    name := uName
    var actionHistory : Lists.List = Lists.create()
    ...
  def getName() : String = name
    ...


resource module DocumentLock
```

```
  def init(uName : String) : Unit
    name := uName
    var actionHistory : Lists.List = Lists.create()
    ...
  def getName() : String = name
    ...


resource module DocumentLock
  require SigUserInfo as uInfo
  def sign() : Boolean =
    var signee : String = uInfo.getName()
    ...
  ...

resource module Main
  instantiate UserInfo() as uInfo
  instantiate DocumentLock(uInfo) as docLock
  ... // client code
```

I/O Capabilities

```
resource module Main
  require FFI
  instantiate FileIO(FFI)
  instantiate Logger(FileIO)
  instantiate Client(Logger)
```

The OS passes a foreign function interface (FFI) capability to Main

The FFI capability is used to instantiate the I/O module

We can restrict the FileIO capability by implementing a logging facility on top of it

The Client can write information to the log, but assuming Logger is implemented securely, it cannot do any other File I/O.

Note: the security of Logger can be verified simply by inspecting the Logger type!

Capability-Safe Modules: Discussion

- Resource modules are like Newspeak, or Units
  - Can only be instantiated or passed as parameters
  - Syntax as convenient as Java import from within a module
  - Slightly less convenient for clients that must instantiate/pass resource modules—but permits more reasoning in exchange

- Pure modules are unrestricted, as in Java
  - Hopefully lower cost overall relative to Newspeak/Units

- Main can require the foreign function interface (FFI)
  - It then passes the FFI capability to I/O modules
  - Shortcut: also OK for main to require modules that take only the FFI as a parameter

Demo

A Capability-Safe Object Calculus

$$
\begin{array}{lcl}
e & ::= & x \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e \\
 & | & \texttt{bind}\ x = e\ \texttt{in}\ e \\
s & ::= & \texttt{stateful} \mid \texttt{pure}
\end{array}
\qquad
\begin{array}{lcl}
d & ::= & \texttt{def}\ m(x : \tau) : \tau = e \\
 & | & \texttt{var}\ f : \tau = x \\
\tau & ::= & \{\overline{\sigma}\}_s \\
\sigma & ::= & \texttt{def}\ m : \tau \to \tau \\
 & | & \texttt{var}\ f : \tau
\end{array}
$$

- Calculus includes objects, methods, mutable fields
- Structural object types
  - `stateful` (have mutable fields/capture state) or `pure`
  - A `stateful` type is a supertype of the equivalent `pure` type
- A `bind` construct for module translation
  - restricts the environment of the second expression to contain only the variable x – *cf.* Scala's Spores [MHO14]

A Capability-Safe Object Calculus

$$\Gamma_{stateful} = \{x : \{\overline{\sigma}\}_{\texttt{stateful}} \mid x : \{\overline{\sigma}\}_{\texttt{stateful}} \in \Gamma\}$$

$$\frac{\Gamma_{pure} = \Gamma \setminus \Gamma_{stateful} \qquad \Gamma_{pure},\ y : \tau_1 \mid \Sigma \vdash^{z} e : \tau_2}{\Gamma \mid \Sigma \vdash_{\texttt{pure}} \texttt{def}\ m(y : \tau_1) : \tau_2 = e \ : \ \texttt{def}\ m : \tau_1 \to \tau_2} \ (\textsc{DT-DefPure})$$

$$\frac{\Gamma,\ x : \tau_1 \mid \Sigma \vdash^{z} e : \tau_2}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{stateful}} \texttt{def}\ m(x : \tau_1) : \tau_2 = e \ : \ \texttt{def}\ m : \tau_1 \to \tau_2} \ (\textsc{DT-DefStateful})$$

$$\frac{\Gamma \mid \Sigma \vdash^{z} x : \tau}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{stateful}} \texttt{var}\ f : \tau = x \ : \ \texttt{var}\ f : \tau} \ (\textsc{DT-VarX})$$

- Key rule DT-DefPure removes `stateful` variables from the context when checking a `pure` method
- DT-VarX is only valid in `stateful` objects

Translating Modules to the Calculus

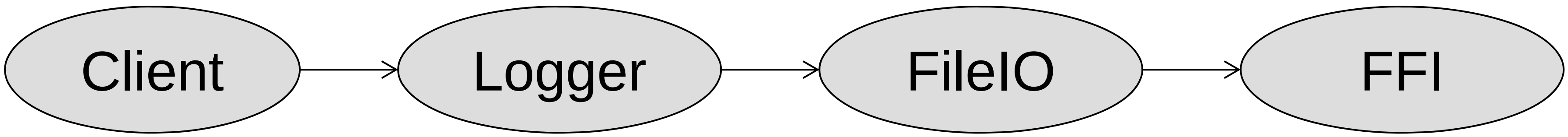
```
resource module Main
  require FFI
  instantiate FileIO(FFI)
  instantiate Logger(FileIO)
  instantiate Client(Logger)
  ...
```

```
def Main(ffi:FFI):Tmain
  bind
    ffi = ffi
    FileIO = FileIO
    Logger = Logger
    Client = Client
  in let
    fio = FileIO(ffi)
    log = Logger(fio)
    client = Client(log)
  in ...
```

- Modules with require become functions (cf. functors)
- bind is used to bind only the imported modules
- instantiate appropriate modules by applying functions

Note: some details simplified for presentation

The Nature of Authority



- What is the authority of Client?
  - given reasonable implementations of Logger/FileIO abstractions
- Prior work's answer: Logger, FileIO, and FFI
- But we argue that Client can only log
  - It cannot do arbitrary File I/O, let alone call arbitrary foreign functions
  - Authority should be viewed as non-transitive
    - There is File IO going on, but it is being done by the Logger
    - Client's authority is only to Logger, unless/until Logger returns a FileIO reference to the Client
  - Enables reasoning about *authority restriction*
    - Logger restricts the FileIO capability to only support logging
    - Future work: additional type system support for this

Authority Safety

- Definition of Authority: the objects I can access directly
  - In my fields
  - Captured in the scope of my methods
- Theorem [Authority Safety]: the authority of an object o increases (by adding an object v) only when:
  - o creates a new object value v
  - A method of o is invoked, passing an argument value v
  - A method that o invoked returns, returning a value v
- Practical consequence:
  - Can reason about an object's authority via calls to its interface
  - **Modules are objects**, so this applies to modules, too
    - Nothing special is needed to handle **dynamically loaded modules**

Type Tests and Capability Safety

```
class BaseLogger
  def log(s:String)
```

```
class ExposedLogger
    extends BaseLogger
  def getLogFile():File
```

```
if (log instanceof ExposedLogger)
  ((ExposedLogger) log).getLogFile().delete()
```

- Would like interface to restrict operations we can perform
  - But downcasts are a problem
- Wyvern's design
  - Structural types: no downcasts possible
  - Datatypes: fixed set of subtypes
    - Pattern matching is OK – can enumerate all possibilities
  - Open tagged types [LASP15]
    - Also allow pattern matching downcasts
    - Lose reasoning about interface—but only when this construct is used
    - Contrast Java – every non-`final` type is open and tagged

Capability-Safe Reflection

```
val m:ObjectMirror = reflect(baseLogger)
val log:ObjectMirror = m.invoke(“getLogFile”)
log.invoke(“delete”)
```

- Reflection can potentially violate capability safety
  - Above: can invoke hidden method on baseLogger
- Safe reflection in Wyvern – universally available
  - Only provides access to methods visible when the mirror was obtained
  - Can’t do anything with reflection that you can’t do without it
  - A reflection capability can be restricted to a narrower type
- Unsafe reflection also provided
  - Access all members – useful for debugging
  - This reflection is available only as a resource module
    - Thus subject to the architectural control mechanisms described above

Capability-Based Architectural Control

How can I enforce key architecture properties?

- **Own your architecture**
  - Architecture specification under source control
  - Use capabilities to delegate resource access in a limited way
- **Use a capability-safe language**
  - Treat resource modules as capabilities – distinct in type system
  - Non-transitive authority for capability restriction
  - Design of type tests and reflection enhances type-based reasoning

Coming your way soon as part of the Wyvern project

- Thanks: NSA Lablet, DARPA BRASS program