Secure Programming via Libraries

A library for information-flow in Haskell

Alejandro Russo (russo@chalmers.se)

Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

Encoding information-flow in Haskell

[Li, Zdancewic 06]

- Show that it is possible to guarantee IFC by a library
- Implementation in Haskell using Arrows [Hughes 98]
- Arrows? A generalization of Monads [Wadler 01]
- Pure values only
  - No side-effects
- One security label for data
  - All secret or all public!

Encoding information-flow in Haskell

[Tsai, Russo, Hughes 07]

- Extend the library by Li and Zdancewic
  - More than one security label for data
  - Concurrency
- Major changes in the library
  - New arrows
  - Lack of arrow notation
- Why arrows?
  - Li and Zdancewic argue that *monads are not suitable for the design of such a library*

A lightweight library for Information-flow in Haskell

[Russo, Claessen, Hughes 08]

- Lightweight
  - Approximately 325 lines of code
  - Static type-system of Haskell to enforce non-interference
  - Dynamic checks when declassification occurs
- Use Monads (not Arrows!)
  - Programmers are more familiar with Monads than Arrows

A lightweight library for Information-flow in Haskell

[Russo, Claessen, Hughes 08]

- The library **relies on** Haskell
  - Capabilities to maintain abstraction of data types
    - Haskell module system
  - Haskell is **strongly typed**
    - We cannot cheat!

```
unsafePerformIO :: IO a -> a
unsafeCoerce :: a -> b
```

- There are extensions of Haskell that break these two requirements!
- For a full list, please visit the proposal of SafeHaskell
  - An extension of Haskell to disallow those dangerous features than can jeopardize security
  - Join work with David Mazieres et al. at Stanford university.

Why Haskell?

- Clear separation of pure computations with those with side-effects
- Every computation with side-effects is encapsulated into the IO monad
- Side-effects can encode information about secret data
- It is necessary to control them
  - It is known where they occur! Just look at the type!

Side-effects and IO

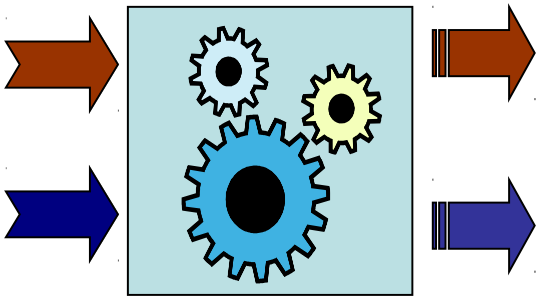
- Just look at the type!

```
f1 :: Eq a => a -> [a] -> ([a], Bool)

f2 :: (Show a, Eq a) => Int -> a -> ([a], IO Bool)
```

- All bets are off if an IO computation comes from untrustworthy code
  - It is not known the side-effects that it will produce

```
f1 x xs = (take 10 (cycle xs), elem x xs)

f2 n x = (take n (iterate id x),
          do putStrLn "Hi!"
             putStrLn "The arguments of the function are"
             putStrLn $ "x = " ++ show x
             putStrLn $ "n = " ++ show n
             return True )
```

Secure Pure Computations



```
f :: (Char {- secret -}, Int)
   -> (Char {- secret -}, Int)
```

```
f (c, i) = ( chr(ord c + i), i)
```

YES

```
f (c, i) = (chr(ord c + i), ord c)
```

NO

```
f (c, i) = (chr(ord c + 1), i+1)
```

YES

```
f (c, i) | c > 65      =  (c, 42)
         | otherwise  =  (c, i)
```

NO

A Security Monad for Pure Computations

- Security monad
  - It assigns a security level to data
  - Once inside the monad, it is not possible to escape!

```
data Sec s a -- abstract
instance Monad (Sec s)
```

  - We represent security levels by singleton types

```
secret :: Sec H Int
secret = ...
```

H

```
known :: Sec L Int
known = ...
```

L

Using Sec

```
f :: (Char {- secret -}, Int)
    -> (Char {- secret -}, Int)

f' :: (Sec H Char, Int)
    -> (Sec H Char, Int)
```

YES

```
f (c, i) = ( chr(ord c + i), i)
```

YES

```
f' (sec_c, i) = (do c <- sec_c
                    return (chr(ord c + i))
                , i)
```

Using Sec

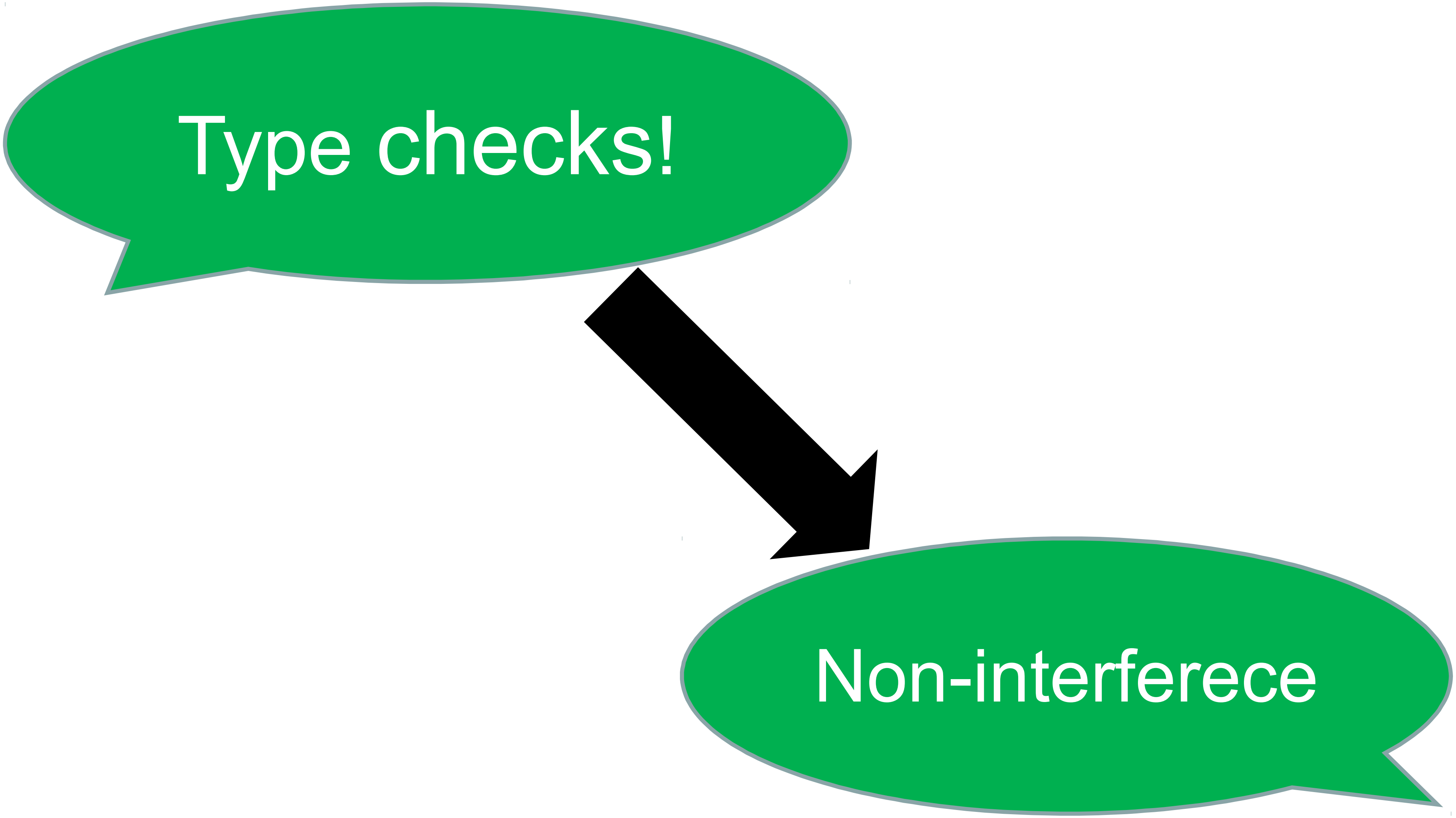
```haskell
f :: (Char {- secret -}, Int)
    -> (Char {- secret -}, Int)

f' :: (Sec H Char, Int)
     -> (Sec H Char, Int)

f (c, i) = (chr(ord c + i), ord c)
```

NO

```haskell
f' (sec_c, i) = ( do c <- sec_c
                     return (chr(ord c + i))
                ,do c <- sec_c
                     return (ord c) )
```

NO

Security Guarantee

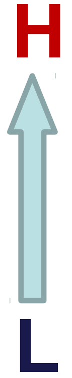


A Security Monad for Pure Computations

- Security monad
  - It assigns a security level to data
  - Once inside the monad, it is not possible to escape!

```
data Sec s a -- abstract
instance Monad (Sec s)
```

  - We represent security levels by singleton types
  - What about the security lattice?



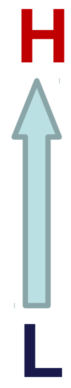
Security Lattice

- We model it using type classes in Haskell
  - Constrains to polymorphic types

```haskell
class Less s s' where
      less :: s -> s' -> ()
```

  - Encoding two-point lattice is just provide instances for that type class

```haskell
instance Less L H where
   less _ _ = ()

instance Less L L where
   less _ _ = ()

instance Less H H where
   less _ _ = ()
```



Security Monad and the Security Lattice

- Push up information in the security lattice

```
up :: Less s s' => Sec s a -> Sec s' a
```

- It allows to convert public values into secrets

```
fup :: Sec L Int -> Sec H Char
fup sec_i = do i <- up (sec_i)
               return (chr i)
```

- What if it is possible to make the following instance?

```
instance Less H L where
   less _ _ = ()
```

Security Monad and the Security Lattice

- The library works as long as
  - Attackers cannot define method `less` for arbitrary instances of the type class `Less`
- How to ensure that?
  - Mainly by the abstraction power of Haskell's module system

Arquitecture

```
module X where

import SecLib.Untrustworthy
import SecLib.LatticeLH

...
```

Importing SecLib.Trustworthy

- `SecLib.Trustworthy` must not be imported by untrustworthy code
  - Otherwise, no security guarantees are possible

```
instance Less H L where
  less _ _ = ()
```

Other Assumptions

- The monad `Sec s` must remain abstract
  - Guarantee by the installation of the library
  - `Sec.hs` is not an exposed module
- Use of unsafe Haskell extensions
  - `StandaloneDeriving`
  - `System.IO.Unsafe`
    - `unsafePerformIO, unsafeIterleaveIO, etc.`
  - `OverlappingInstances`
- `Check SafeHaskell (work-in-progress)`
  - `A Haskell extension to safely execute untrusted Haskell code`

Security API for Pure Computations

```
data Sec s a -- abstract
instance Monad (Sec s)


up :: Less s s' => Sec s a -> Sec s' a


module X where

import SecLib.Untrustworthy
import SecLib.LatticeLH
```