Secure Programming via Libraries (T3)
ECI 2011 - Day 4

Alejandro Russo

Chalmers University of Technology

Exercise 1 DCLabels

Verify the following assertions using the module `DCLabel`. In other words, implement the following DCLabels in Haskell and use the function `canflowto` to verify that the relationship $\sqsubseteq$ holds. Justify why it holds (or why it does not).

1. $\{[Alice], [Alice]\} \sqsubseteq \{[Alice \wedge Bob], [Alice]\}$
2. $\{[Alice], [Alice]\} \sqsubseteq \{[\,], [Alice]\}$
3. $\{[Alice], [\,]\} \sqsubseteq \{[\,], [Alice]\}$
4. $\{[Alice \vee Bob], [Deain]\} \sqsubseteq \{[Alice] \wedge [Bob], [Alice \vee Deain]\}$
5. $\{[\,], All\} \sqsubseteq \{All, [\,]\}$

Exercise 2 DCLabels and privileges

As the exercise above, verify the following assertions using the module `DCLabel`. In other words, implement the following DCLabels in Haskell and use the function `canflowto_p` to verify that the relationship $\sqsubseteq$ holds. Justify why it holds (or why it does not).

1. $\{[Alice], [Bob]\} \sqsubseteq_{Deain} \{[Deain], [Deain]\}$
2. $\{[Alice], [Bob]\} \sqsubseteq_{Alice} \{[Deain], [Deain]\}$
3. $\{[Alice], [Deain]\} \sqsubseteq_{Alice} \{[Deain], [Deain]\}$
4. $\{[Alice \vee Bob] \wedge Charlie, [\,]\} \sqsubseteq_{Bob} \{[Charlie], [\,]\}$

Exercise 3 `Labeled` values inside `Labeled` values

When writing programs, it might happen that it appears nested `Labeled` values. For instance, a program might be manipulating values of type `Labeled l (Labeled l a)`, `Labeled l (Labeled l a, Labeled l a)`, and so on. It is sometime useful to remove such nested levels if possible. For each simplification below, justify why the DCLabel of the result makes sense.

1. Implement the following function

   ```
   simplify :: LIO DCLabel () (Labeled DCLabel (Labeled DCLabel String))
               → LIO DCLabel () (Labeled DCLabel String)
   ```

   that takes two nested `Labeled` and returns one. To implement this function, you will need to use `return`, bind, `label`, `unlabel` and `getLabel`. Function `getLabel :: (Label l) ⇒LIO l s l` returns what is the current label of a `LIO` computation. More precisely, if you have a `LIO` computation

   ```
   lio_computatoin = do
                       ....
                       ....
                       l ← getLabel
                       ....
                       ....
   ```

   variable `l` will then contain the current label at the time of calling `getLabel`.
   To try out the function that you implement above, you might consider the following piece of trustworthy code.

```
module Ex4T where

import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB

import Ex4U (simplify)

d1 :: LIO DCLabel () (Labeled DCLabel String)
d1 = label (newDC ("Alice" ∨ "Bob") "Alice") "Some data from Alice"

d2 = do l ← d1
        label (newDC "Deain" "Deain") l

execute = do (result, label) ← evalLIO (do l ← simplify d2
                                           unlabel l) ()
             putStrLn $ "The result is: " ++ show result
             putStrLn $ "With the label: " ++ prettyShow label
```

where `d2` is a labeled value, with label $\{[Deain], [Deain]\}$, that contains the labeled value `d1` with label $\{[Alice \vee Bob], [Alice]\}$.

2. Implement the following function

```
simplify2 :: LIO DCLabel () (Labeled DCLabel String,
                             Labeled DCLabel Int)
          → LIO DCLabel () (Labeled DCLabel (String, Int))
```

that takes a pair of labeled values and returns a labeled pair.
To try out the function that you implement above, you might consider the following piece of trustworthy code

```
module Ex4T2 where

import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB

import Ex4U2 (simplify2)

d1 :: LIO DCLabel () (Labeled DCLabel String)
d1 = label (newDC ("Alice" ∨ "Bob") "Alice") "Some data from Alice"

d2 :: LIO DCLabel () (Labeled DCLabel Int)
d2 = label (newDC "Bob" "Alice") 42

d3 = do l1 ← d1
        l2 ← d2
        return (l1,l2)

execute = do (result, label) ← evalLIO (do p ← simplify2 d3
                                           unlabel p) ()
             putStrLn $ "The result is: " ++ show result
             putStrLn $ "With the label: " ++ prettyShow label
```

where `d3` is a pair that contains the labeled values `d1` and `d2` with labels $\{[Alice \vee Bob], [Alice]\}$ and $\{[Alice], [Bob]\}$, respectively.

Exercise 4 More programming with LIO

1. Similar to what we did for `Sec`, we want to implement a function that takes two booleans and returns the integer that those boolean values represent when interpreted as binary. The most significant bit is the last boolean. More precisely, we want to implement the function

```
binToInt2 :: LIO DCLabel () (Labeled DCLabel Bool) → LIO DCLabel () (Labeled DCLabel Bool)
             → LIO DCLabel () Int
```

To try out the function that you implement above, you might consider the following piece of trustworthy code

```
module Ex4T3 where

import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB

import Ex4U3 (binToInt2)

b1 :: LIO DCLabel () (Labeled DCLabel Bool)
b1 = label (newDC ("Alice") "Alice") True

b2 :: LIO DCLabel () (Labeled DCLabel Bool)
b2 = label (newDC "Bob" "Alice") True

execute = do (result, label) ← evalLIO (binToInt2 b1 b2) ()
             putStrLn $ "The result is: " ++ show result
             putStrLn $ "With the label: " ++ prettyShow label
```

where `b1` and `b2` are two booleans that are converted into an integer.

2. Implement the function

```
binToInt :: LIO DCLabel () [Labeled DCLabel Bool] → LIO DCLabel () Int
```

that takes a list of public booleans and return the number obtaining by interpreting that list as a binary number. The most significant bit is the last element of the list.