Gradually typed DSLs with Deferrable constraints in Haskell

… through the tale of a Haskell information flow control library

Pablo Buiras, Alejandro Russo, Dimitrios Vytiniotis

Information flow control 101

- Non-interference: “sensitive” input does not flow to “public” output

- Formal model:
  a lattice of security labels,
  security labelled data,
  and observational equivalence w.r.t. a label

```
data Label = L | H

ℓ ⊑ H = True
L ⊑ L = True
H ⊑ L = False
```

- Can be enforced statically (e.g. types, static analyses) or dynamically

LIO: Dynamic IFC for Haskell [Stefan et al.]

```
data LIO a
instance Monad LIO

-- data guarded by label
data Labeled a
```

Just a state monad carrying “current label”

1. *Observing (unlabelling) sensitive data increases current label*
2. *Cannot output data below current label*

LIO: Dynamic IFC for Haskell [Stefan et al.]

```
data LIO a
instance Monad LIO

-- data guarded by label
data Labeled a
```

Just a state monad carrying "current label"

1. *Observing (unlabelling) sensitive data increases current label*
2. *Cannot output data below current label*

```
lconcat :: Labeled String -> Labeled String -> LIO String
lconcat lstr1 lstr2 = do  -- Initial current label ℓcur
    str1 <- unlabel lstr1 -- ℓcur'  = ℓcur  ⊔ (labelOf lstr1)
    str2 <- unlabel lstr2 -- ℓcur'' = ℓcur' ⊔ (labelOf lstr2)
    return (str1 ++ str2) -- Final current label ℓcur''
```

Quite a number of dynamic checks/taints …

Question: *Is there a statically checked variant of LIO?*

Quite a number of dynamic checks/taints …

Question: *Is there a statically checked variant of LIO?*

IDEA 1
Use labels at the type level

```
type class Flows l1 l2
instance Flows L L
instance Flows L H
instance Flows H H
type family Join l1 l2 where ...

-- singleton term-level labels
data SLabel (l::Label) where
  L :: SLabel L
  H :: SLabel H

data Labeled (l::Label) a
```

Quite a number of dynamic checks/taints …

Question: *Is there a statically checked variant of LIO?*

IDEA 1
Use labels at the type level

```
type class Flows l1 l2
instance Flows L L
instance Flows L H
instance Flows H H
type family Join l1 l2 where ...

-- singleton term-level labels
data SLabel (l::Label) where
  L :: SLabel L
  H :: SLabel H

data Labeled (l::Label) a
```

IDEA 2
Use a “Hoare state monad”

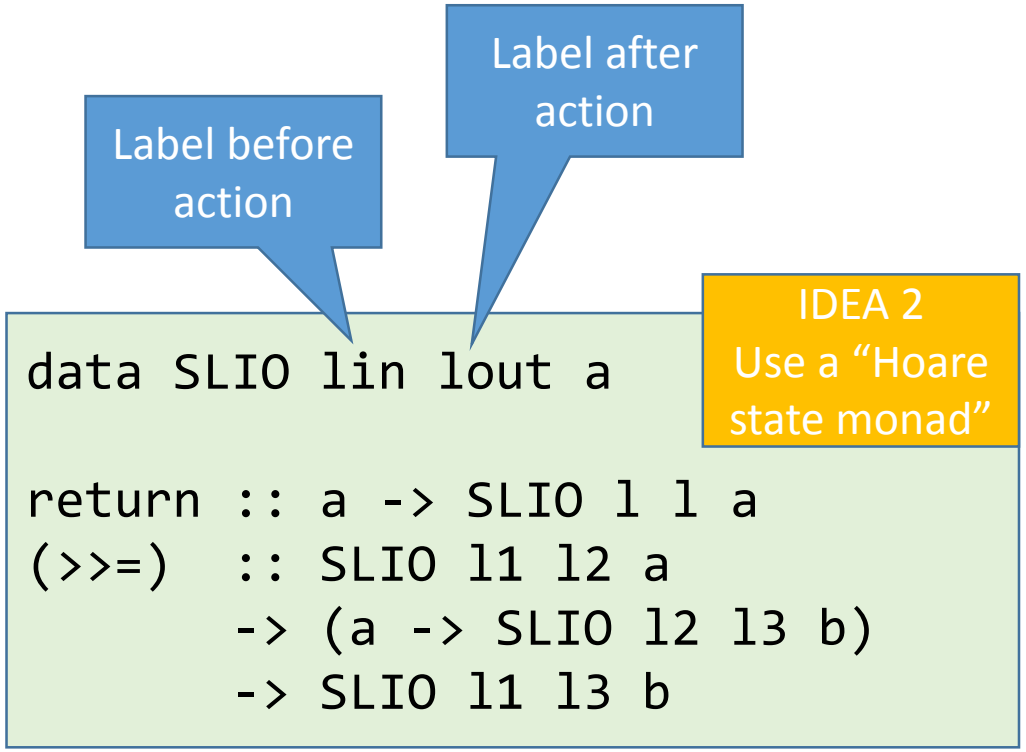
```
data SLIO lin lout a

return :: a -> SLIO l l a
(>>=)  :: SLIO l1 l2 a
       -> (a -> SLIO l2 l3 b)
       -> SLIO l1 l3 b
```

Quite a number of dynamic checks/taints …

Question: *Is there a statically checked variant of LIO?*

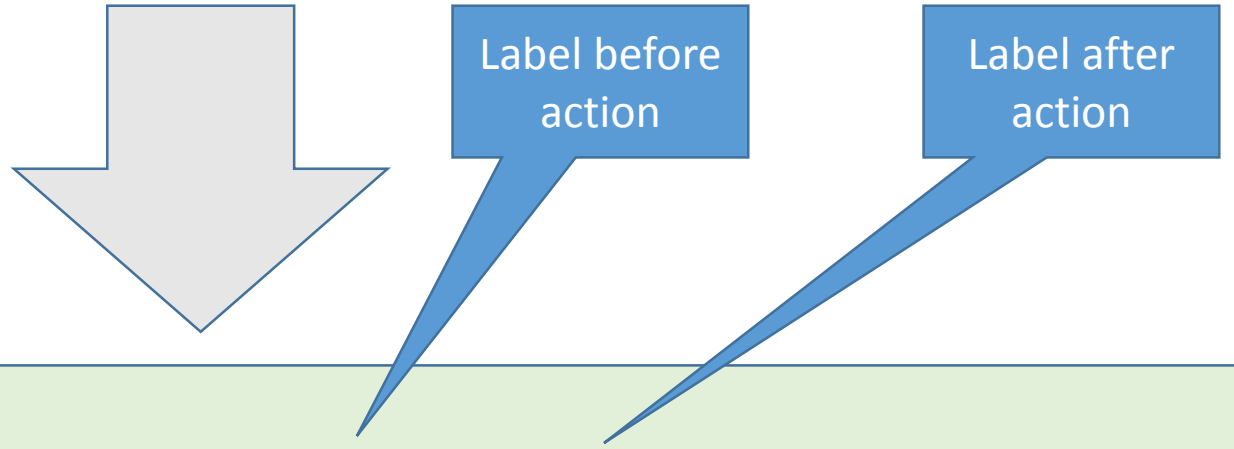
IDEA 1
Use labels at the type level

```
type class Flows l1 l2
instance Flows L L
instance Flows L H
instance Flows H H
type family Join l1 l2 where ...

-- singleton term-level labels
data SLabel (l::Label) where
  L :: SLabel L
  H :: SLabel H

data Labeled (l::Label) a
```



From LIO to SLIO, mechanically

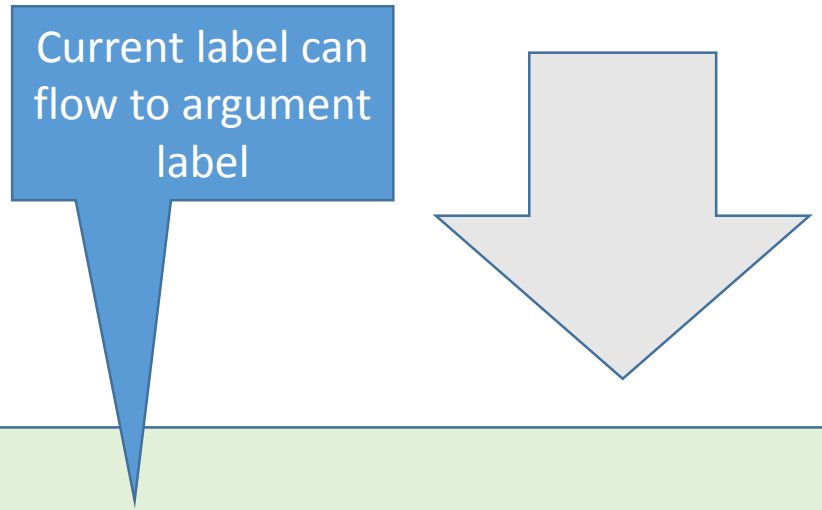
```
unlabel :: Labeled a -> LIO a
-- (1) never fails;
-- (2) taints LIO label with argument label
```




```
unlabel :: Labeled la a -> SLIO l (Join l la) a
```

From LIO to SLIO, mechanically

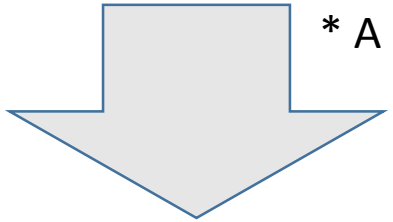
```
label :: Label -> a -> LIO (Labeled a)
-- (1) succeeds only if current label ⊑ argument label;
-- (2) returns labelled data
```




```
label :: Flows l la => SLabel la -> SLIO l l (Labeled la a)
```

From LIO to SLIO; tackling the “label creep”

```
toLabeled :: Label -> LIO a -> LIO (Labeled a)
-- toLabeled ℓ m
-- 1) Check that ℓcur ⊑ ℓ
-- 2) Execute action m starting from ℓcur; new label is ℓcur’
-- 3) Check that ℓcur’ ⊑ ℓ
-- 4) If so, pack the result with label ℓ, final label is ℓcur
```



* A simpler version of

```
toLabeled :: SLIO ℓi ℓo a -> SLIO ℓi ℓi (Labeled ℓo a)
```

Success!

Success!

?

Static types can get complex

```
-- Send a string to a remote server who may leak something
report :: Flows l L => String -> SLIO l l ()




lReport ls1 ls2 = do
   v1 <- unlabel ls1
   v2 <- unlabel ls2
   let res = v1 ++ v2
   report res
   return res
```

Static types can get complex

```
-- Send a string to a remote server who may leak something
report :: Flows l L => String -> SLIO l l ()

lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
lReport ls1 ls2 = do
   v1 <- unlabel ls1
   v2 <- unlabel ls2
   let res = v1 ++ v2
   report res
   return res
```

Static types can get complex

```
-- Send a string to a remote server who may leak something
report :: Flows l L => String -> SLIO l l ()

lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
lReport ls1 ls2 = do
   v1 <- unlabel ls1
   v2 <- unlabel ls2
   let res = v1 ++ v2
   report res
   return res
```

A bit verbose but (depending who you are) may be ok

And its tedious to interact with dynamic data

```
lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
readRemote :: URI -> SLIO li li (∃l. Labeled l String)




readReport = do
   (Exists lv1) <- readRemote “host1”
   (Exists lv2) <- readRemote “host2”
   toLabeled (lReport lv1 lv2)
```

And its tedious to interact with dynamic data

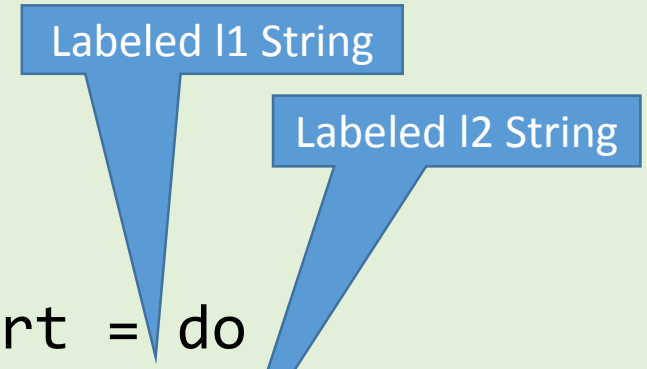
```
lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
readRemote :: URI -> SLIO li li (∃l. Labeled l String)
```

Labeled l1 String

```
readReport = do
   (Exists lv1) <- readRemote “host1”
   (Exists lv2) <- readRemote “host2”
   toLabeled (lReport lv1 lv2)
```

And its tedious to interact with dynamic data

```
lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
readRemote :: URI -> SLIO li li (∃l. Labeled l String)
```




```
readReport = do
   (Exists lv1) <- readRemote “host1”
   (Exists lv2) <- readRemote “host2”
   toLabeled (lReport lv1 lv2)
```

And its tedious to interact with dynamic data

```
lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
readRemote :: URI -> SLIO li li (∃l. Labeled l String)
```

Labeled l1 String

Labeled l2 String

```
readReport = do
   (Exists lv1) <- readRemote “host1”
   (Exists lv2) <- readRemote “host2”
   toLabeled (lReport lv1 lv2)
```

Can you spot the 2 type errors in readReport?

What's wrong with this code?

```
lReport :: Flows (Join (Join ℓi ℓ1) ℓ2) L
        => Labeled ℓ1 String
        -> Labeled ℓ2 String
        -> SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
readRemote :: URI -> SLIO li li (∃l. Labeled l String)
```

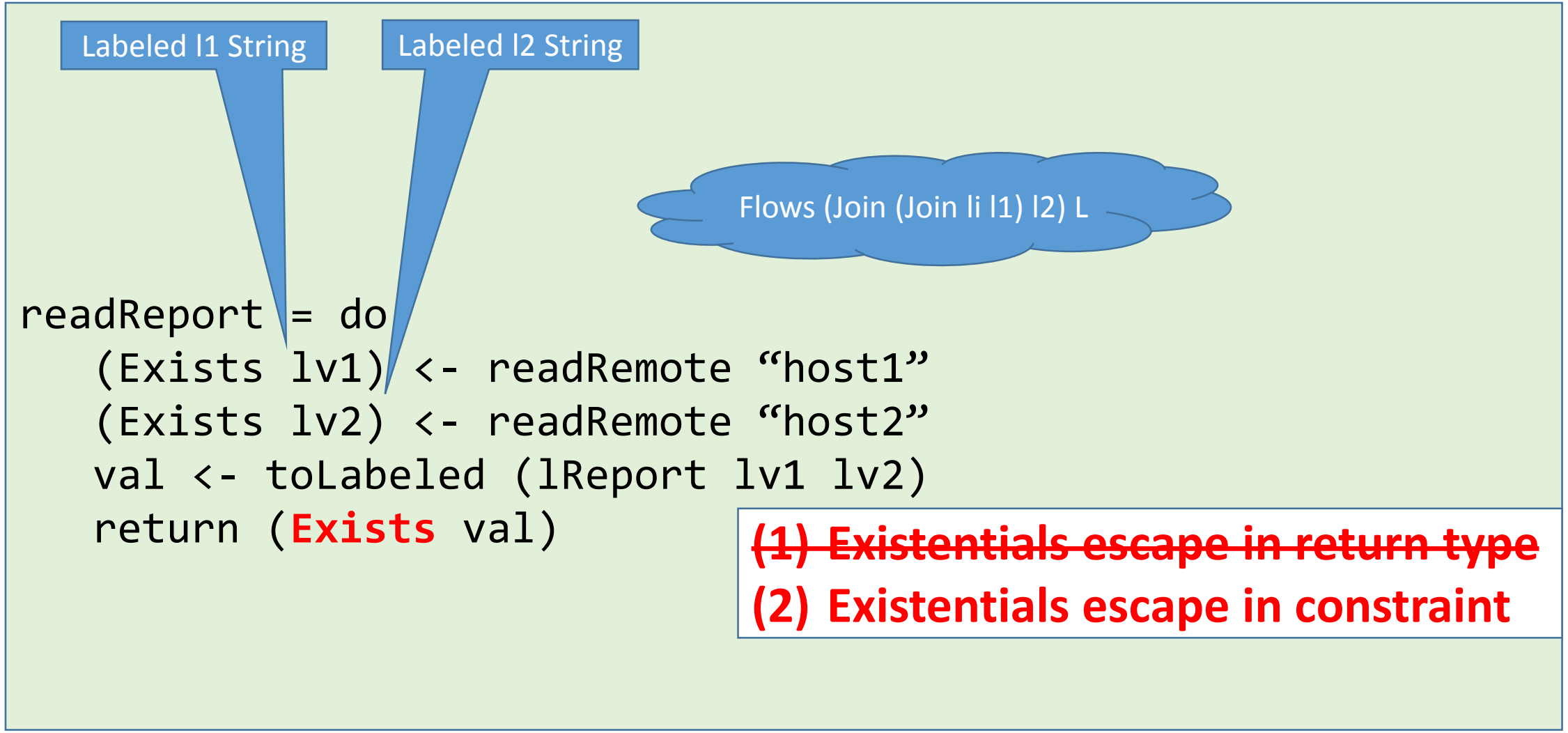
Labeled l1 String

Labeled l2 String

```
readReport = do
   (Exists lv1) <- readRemote
   (Exists lv2) <- readRemote “host2”
   toLabeled (lReport lv1 lv2)
```

(1) Existentials escape in return type
(2) Existentials escape in constraint

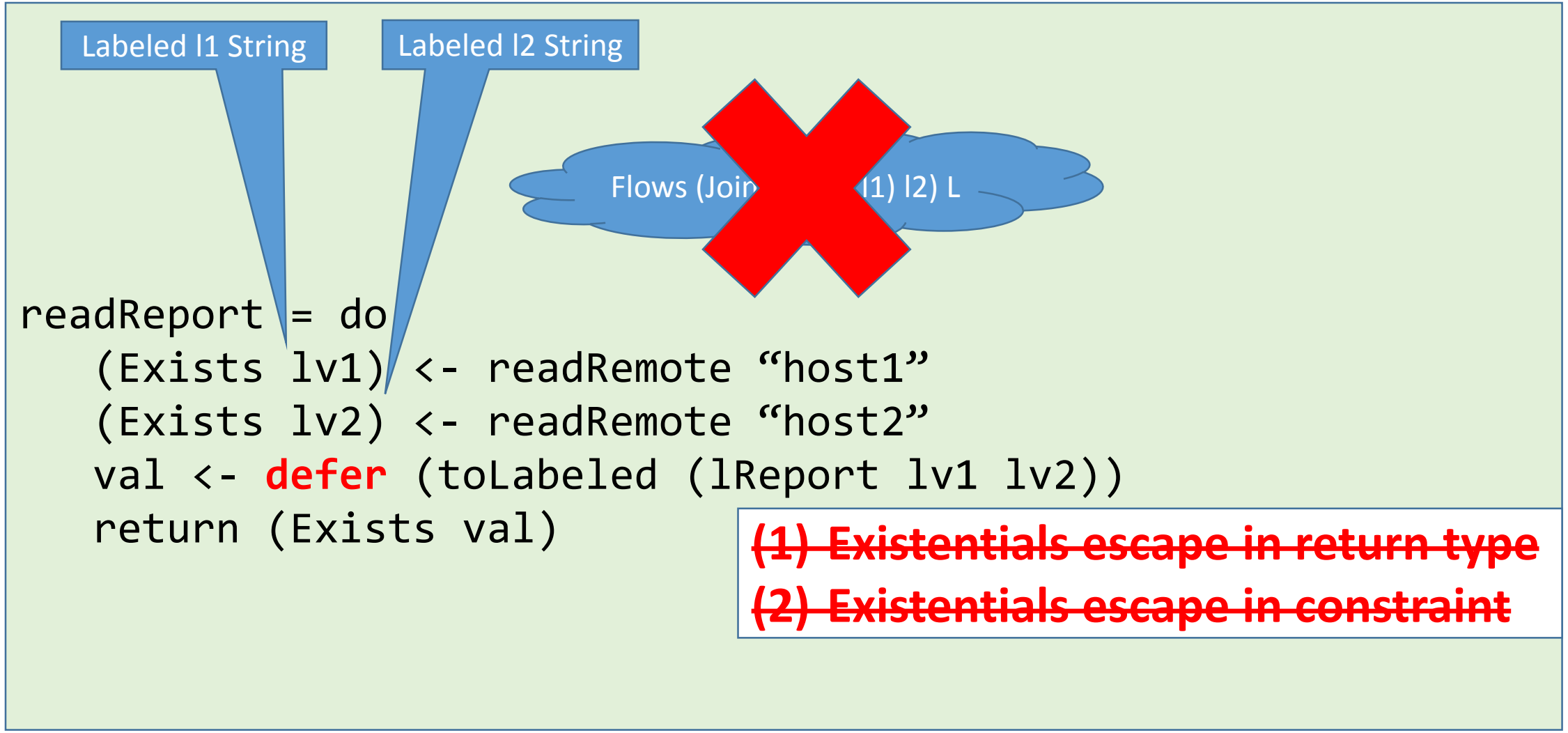
Existential escape in types: easy solution



So, back to dynamic LIO?

Not quite!

What if we could just … defer a constraint?



Gradually typed LIO in Haskell

How to defer a constraint to runtime?

```
deferC :: (c => a) -> a
```

Give me any function that requires constraint c to produce result a, and I will give you back a

Seems seriously implausible! ☺

Key idea: A type class of Deferrable constraints

```
class Deferrable (c :: Constraint) where
   deferC :: Proxy c -> (c => a) -> a
```

Common Haskell-ism to account for the lack of explicit type applications

Easy to give Deferrable *instances*!

```
class Deferrable (c :: Constraint) where
   deferC :: Proxy c -> (c => a) -> a
```

```
-- class providing a singleton
class CLabel l where lab :: Proxy l -> SLabel l

instance (Clabel l1, Clabel l2) => Deferrable (Flows l1 l2) where
   deferC p m = case (lab p1, lab p2) of
     (L,L) -> m
     (L,H) -> m
     (H,H) -> m
     (H,L) -> error "IFC violation!"
     where p1 = ⊥ :: Proxy l1; p2 = ⊥ :: Proxy l2

instance (Deferrable c1, Deferrable c2) => Deferrable (c1,c2)
```

Easy to give Deferrable *instances*!

```
class Deferrable (c :: Constraint) where
   deferC :: Proxy c -> (c => a) -> a
```

```
-- class providing a singleton
class CLabel l where lab :: Proxy l -> SLabel l

instance (Clabel l1, Clabel l2) => Deferrable (Flows l1 l2) where
   deferC p m = case (lab p1, lab p2) of
     (L,L) -> m
     (L,H) -> m
     (H,H) -> m
     (H,L) -> error "IFC violation!"
     where p1 = ⊥ :: Proxy l1; p2 = ⊥ :: Proxy l2

instance (Deferrable c1, Deferrable c2) => Deferrable (c1,c2)
```

GADT pattern matching refines types to cases where we do have instances

That's all the essential ingredients

Read ICFP paper for other technical details

```
data SLIO (c :: Constraint) (li :: Expr Label) (lo :: Expr Label)
labelOf  :: Labeled ℓ a -> SLabel (E ℓ)
getLabel :: HLIO () ℓi ℓi (SLabel (E ℓi ))
unlabel  :: Labeled ℓ a -> HLIO () ℓi (LJoin ℓi ℓ) a
label    :: SLabel ℓ -> a
         -> HLIO (FlowsE ℓi (LVal ℓ)) ℓi ℓi (Labeled (LVal ℓ) a)
toLabeled :: HLIO c ℓi ℓo a -> HLIO c ℓi ℓi (Labeled ℓo a)

defer :: Deferrable c ⇒ HLIO c ℓi ℓo a -> HLIO () ℓi ℓo a
simplify :: c ⇒ HLIO c ℓi ℓo a → HLIO () ℓi ℓo a
```

Expressing constraint as datatype index:
(a) Better control over simplification
(b) Eliminates need for proxy annotations

Distinguishing between label expressions and label values

Demo and thanks!