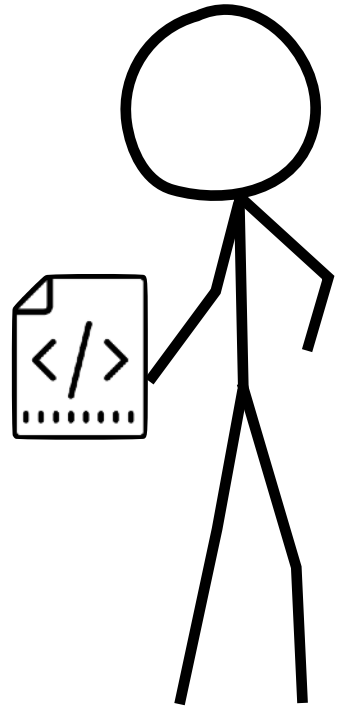
A DEPENDENTLY TYPED LIBRARY FOR STATIC INFORMATION-FLOW CONTROL IN IDRIS

Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

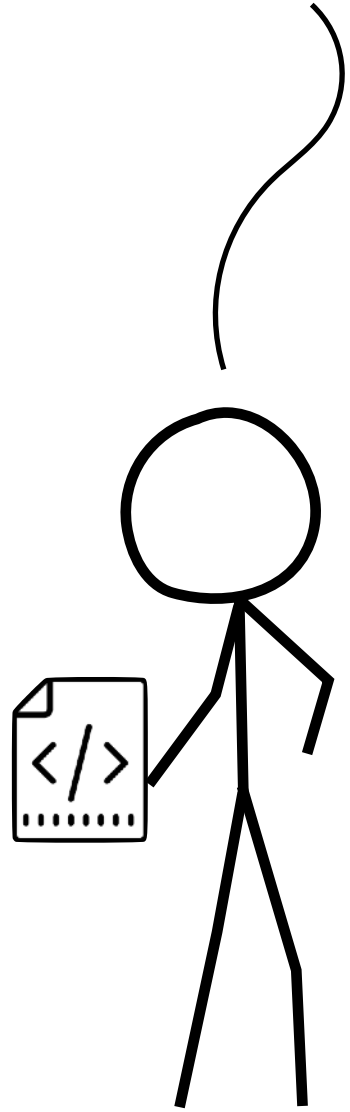
Hmm... I need a library!

Hmm... I need a library!



Hmm... I need a library!

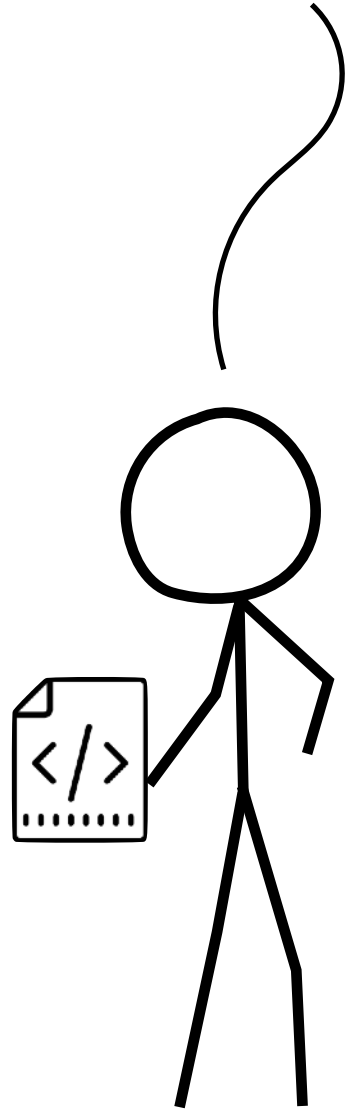
I got a library!



Hmm... I need a library!

But is it secure?

I got a library!

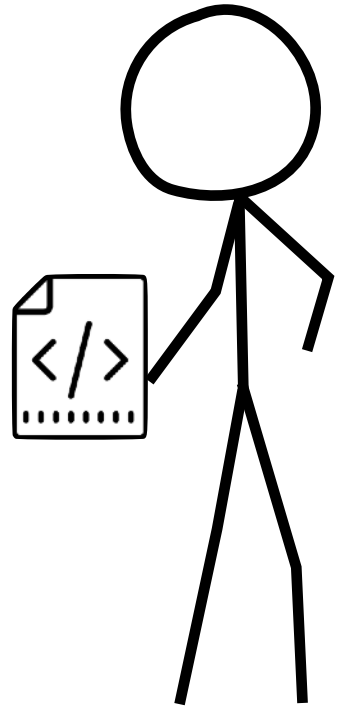


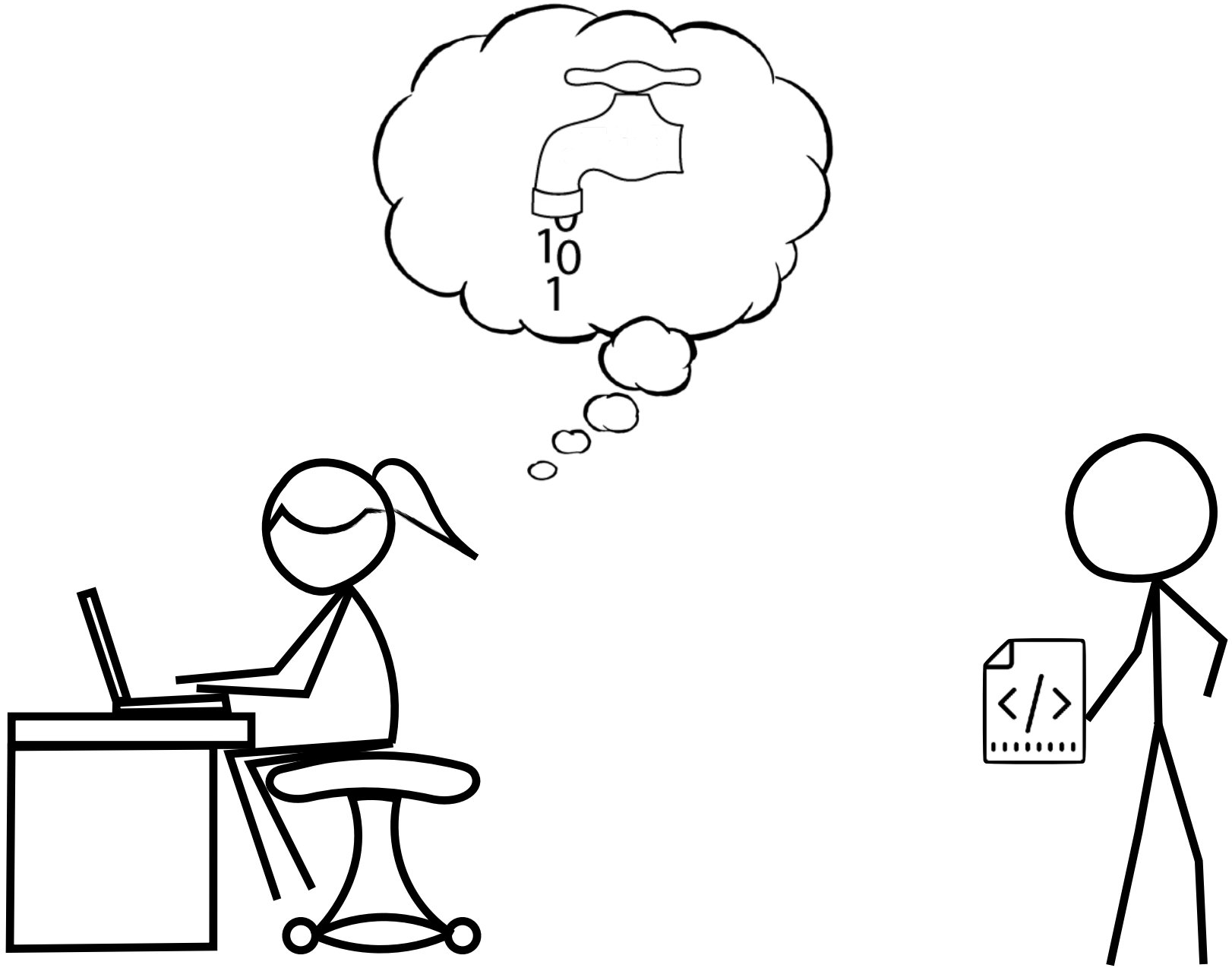
Hmm... I need a library!

But is it secure?

I got a library!

Of course!




10
1
</>


Jif
1
0
1

Paragon

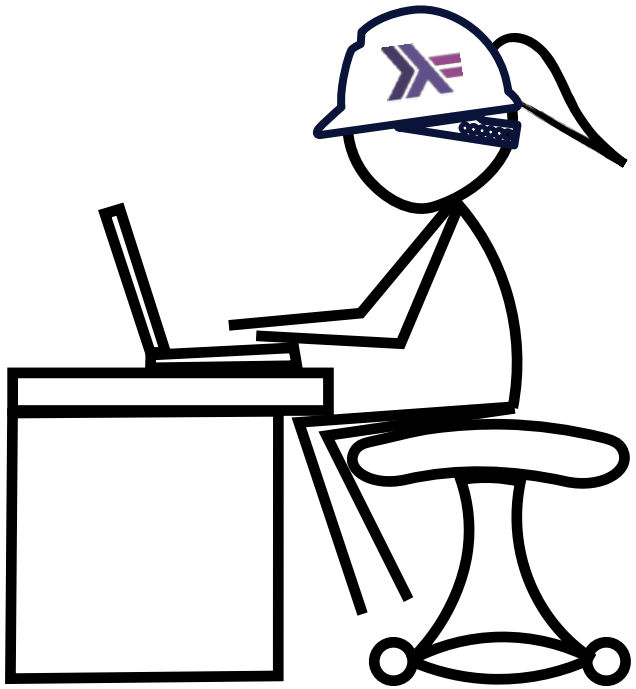
Paragon

Flow Caml

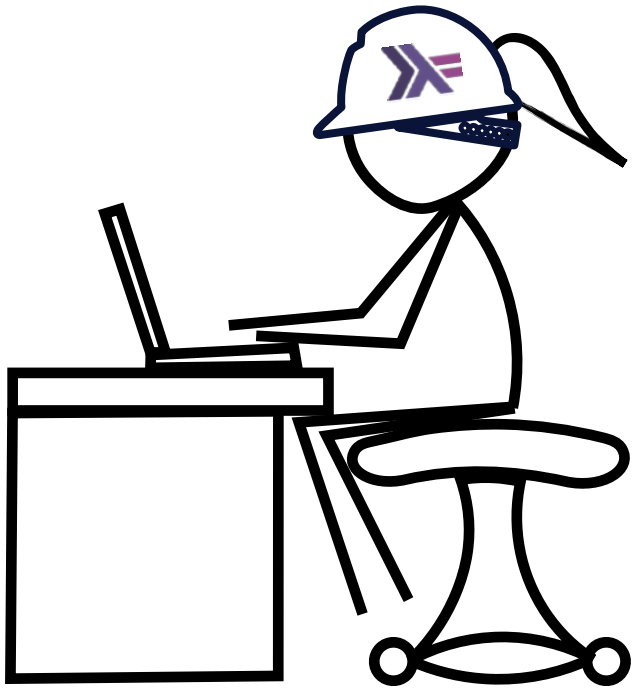
Paragon

Flow Caml

That's a lot of work!

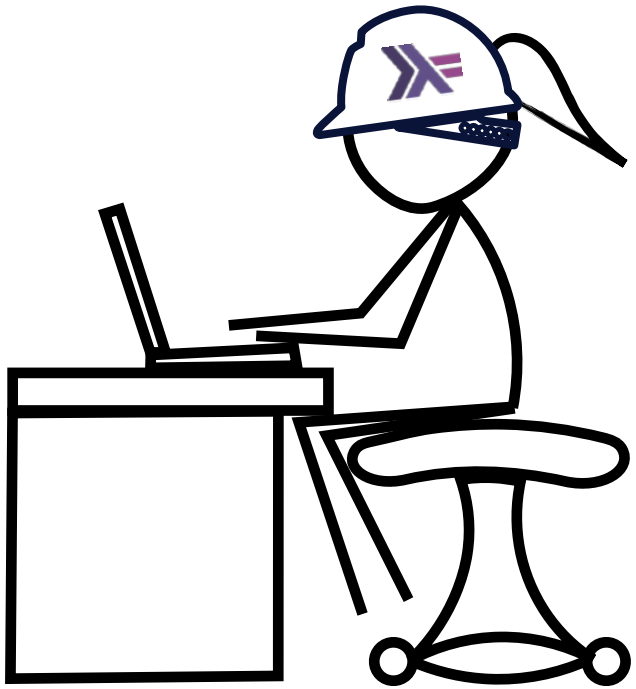


HASKELL LIBRARIES



HASKELL LIBRARIES

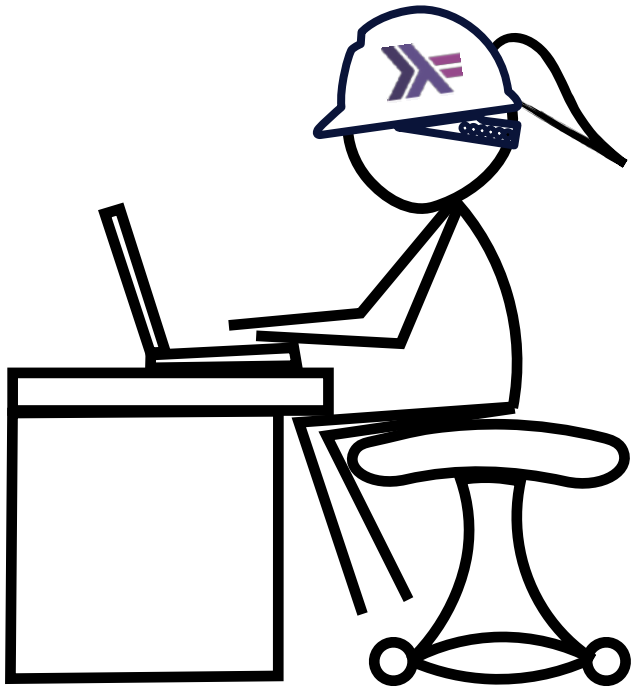
SecLib (Russo et al., 2008)



HASKELL LIBRARIES

SecLib (Russo et al., 2008)

LIO (Stefan et al., 2011)

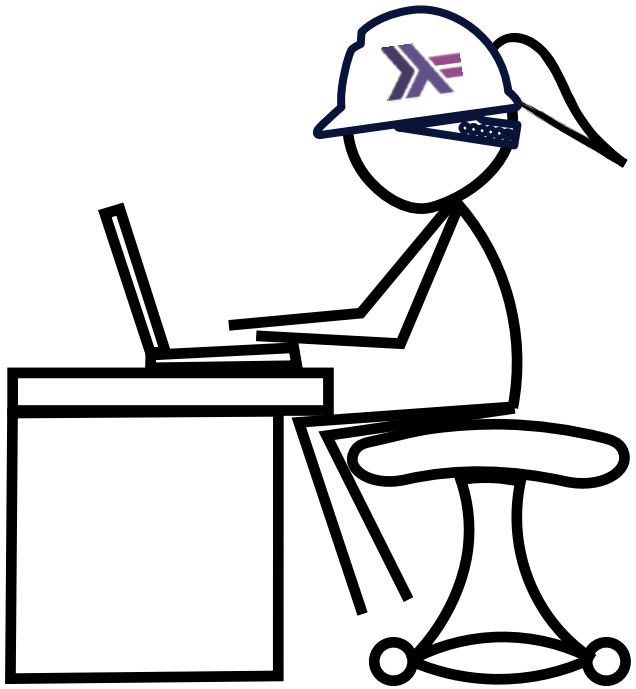


HASKELL LIBRARIES

SecLib (Russo et al., 2008)

HLIO (Buiras et al., 2015)

LIO (Stefan et al., 2011)



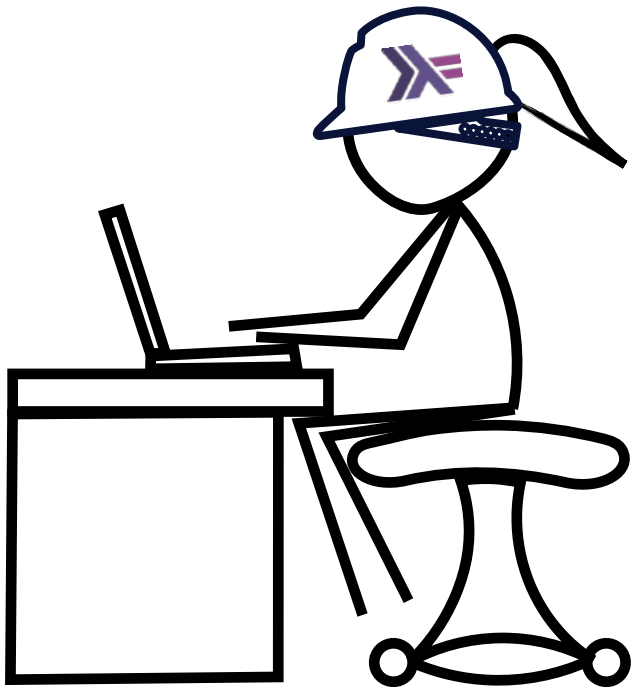
HASKELL LIBRARIES

SecLib (Russo et al., 2008)

HLIO (Buiras et al., 2015)

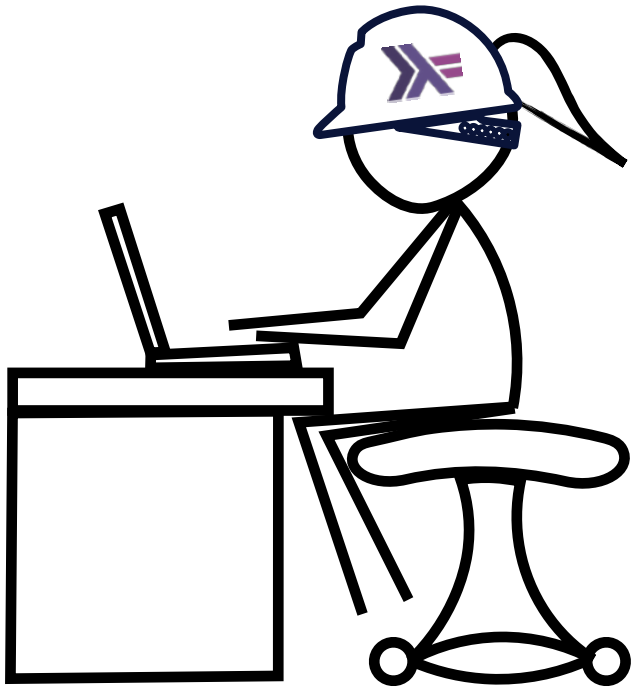
LIO (Stefan et al., 2011)

MAC (Russo, 2015)



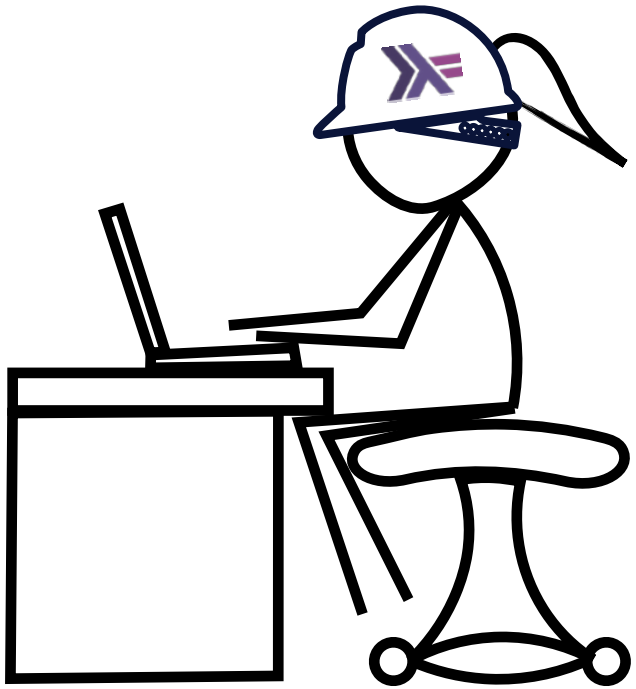
MAC (Russo, 2015)

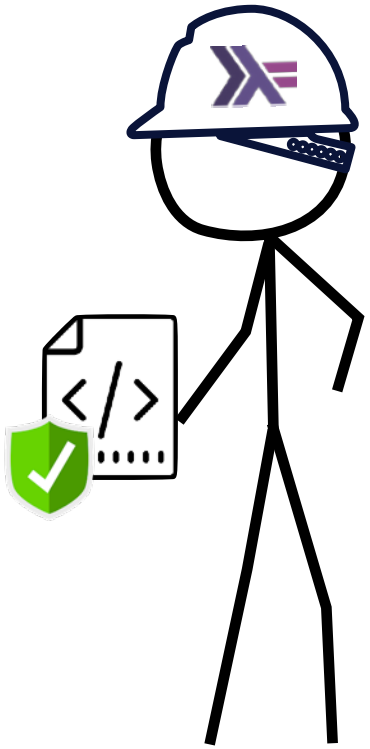
- A lightweight library
- No run-time checks
- Illegal flows rejected at compile-time



MAC (Russo, 2015)

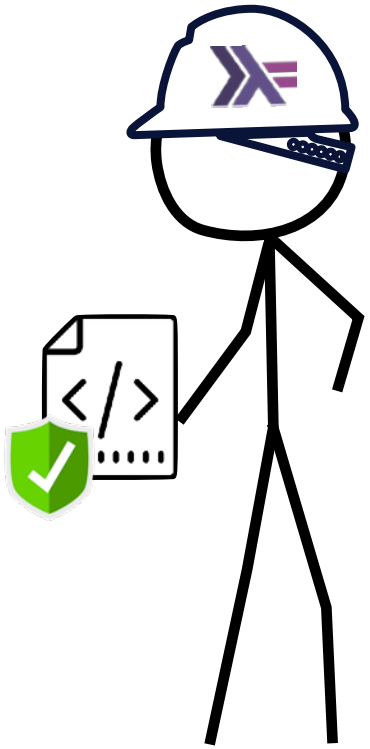
- A lightweight library
- No run-time checks
- Illegal flows rejected at compile-time





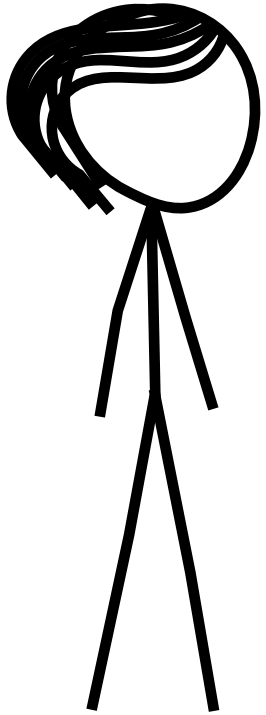
MAC (Russo, 2015)

- A lightweight library
- No run-time checks
- Illegal flows rejected at compile-time
- **Limitation:** statically defined security compartments

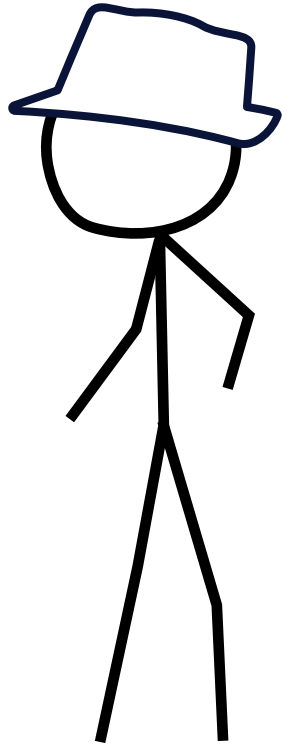


EXAMPLE: AUCTION

Carol

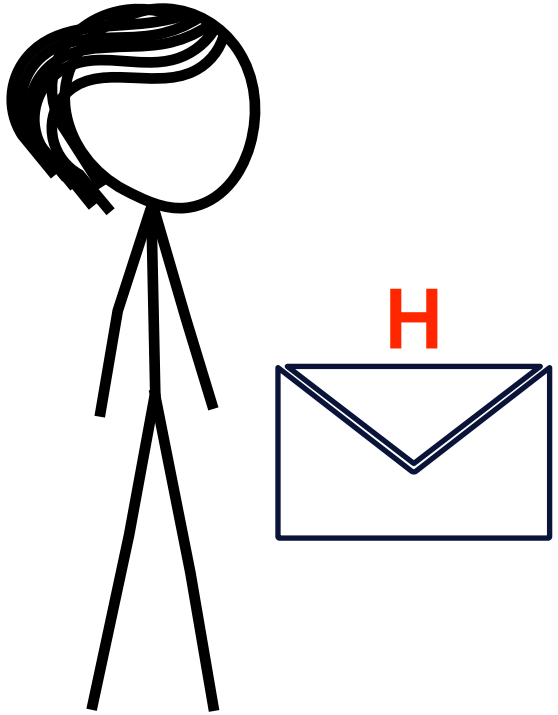


Dan

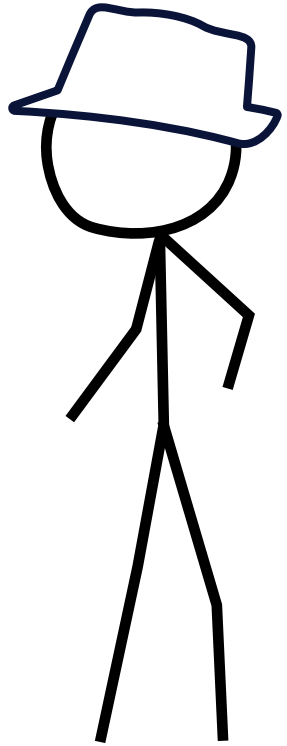


EXAMPLE: AUCTION

Carol

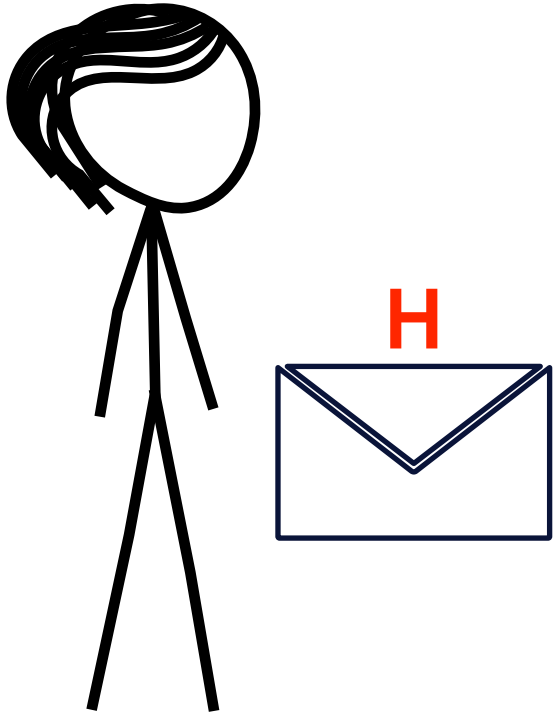


Dan

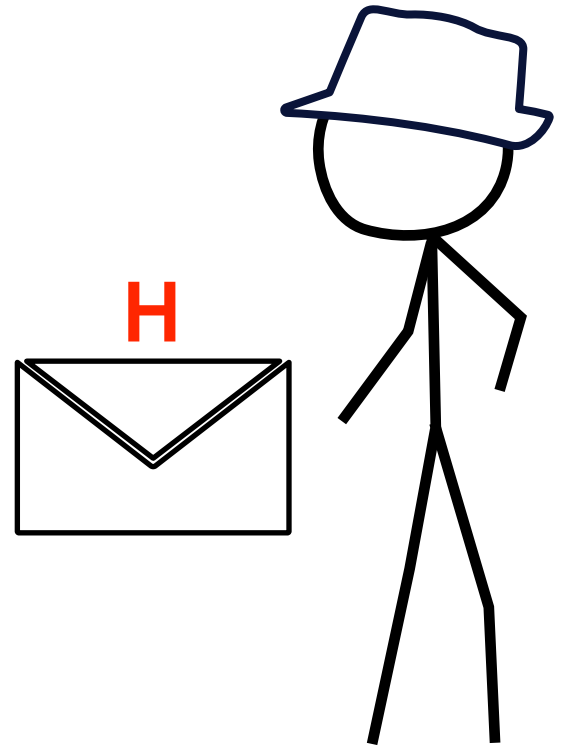


EXAMPLE: AUCTION

Carol



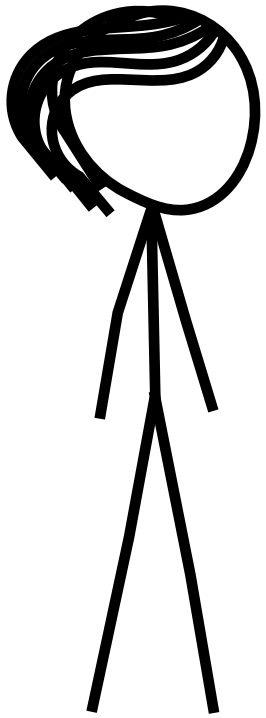
Dan



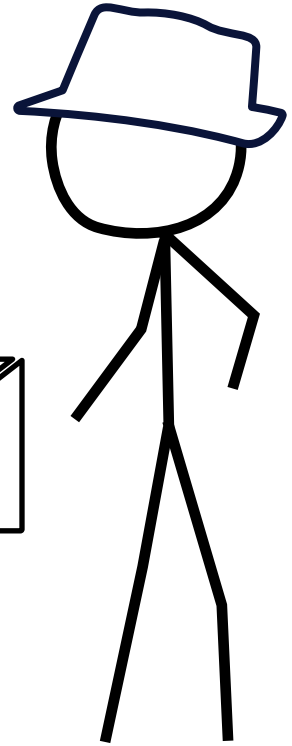
EXAMPLE: AUCTION

H

Carol

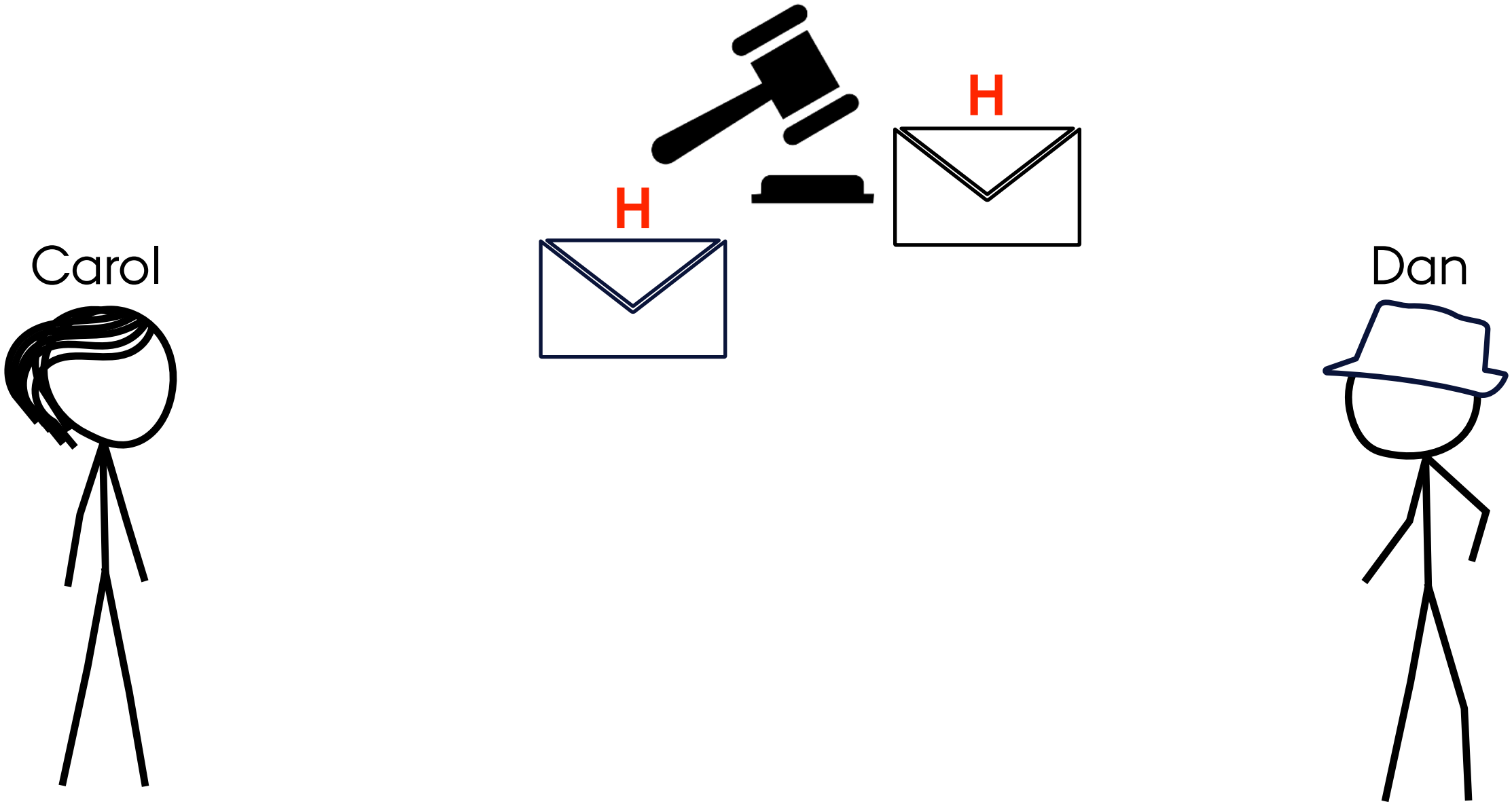


Dan



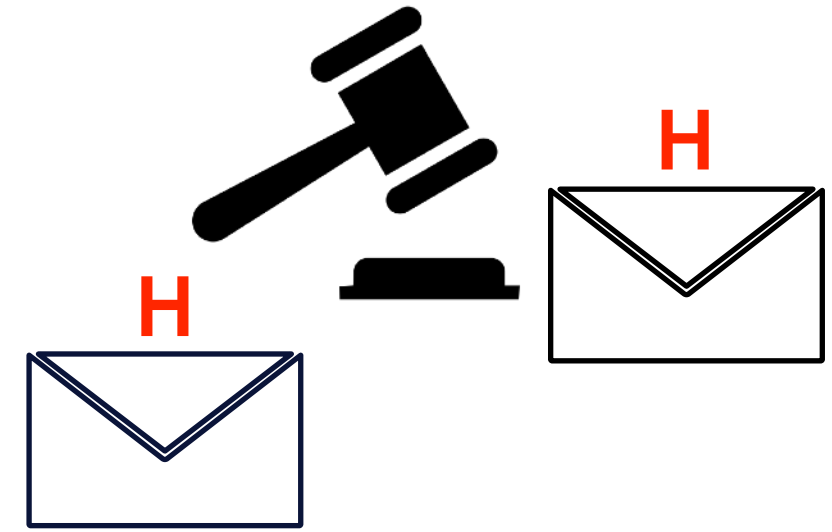
H

EXAMPLE: AUCTION

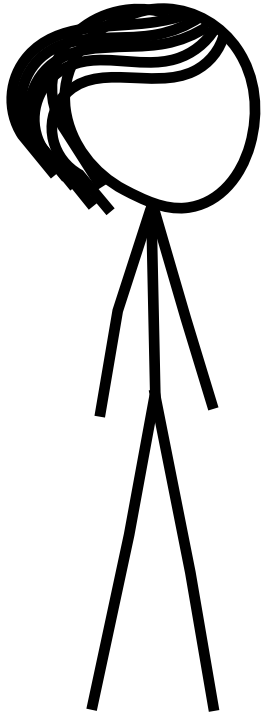


EXAMPLE: AUCTION

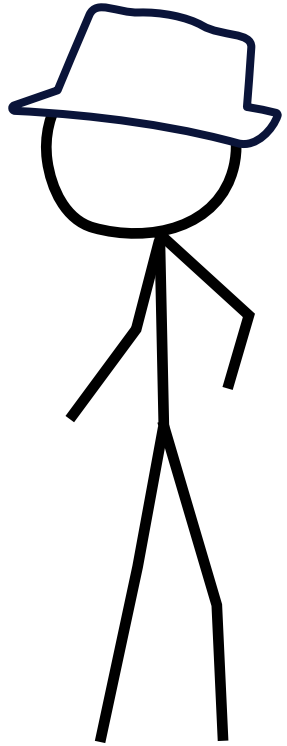
$H \not\subseteq L$



Carol



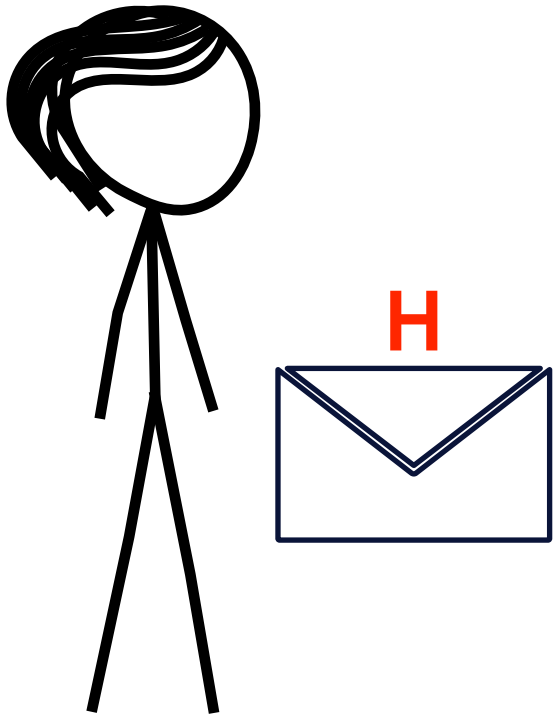
Dan



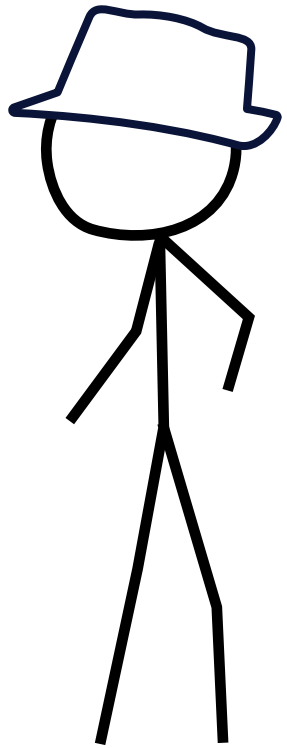
EXAMPLE: AUCTION

H ⊈ L

Carol



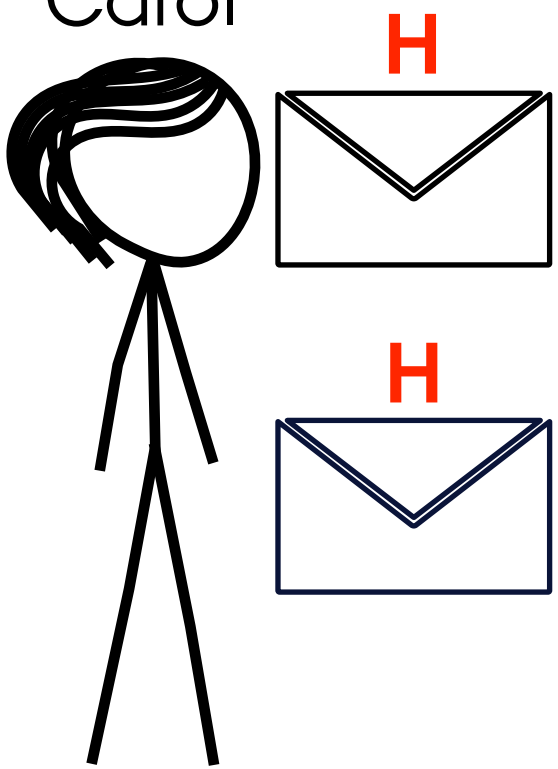
Dan



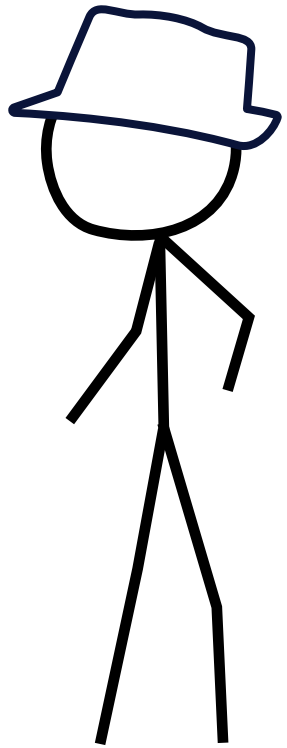
EXAMPLE: AUCTION

$H \not\subseteq L$

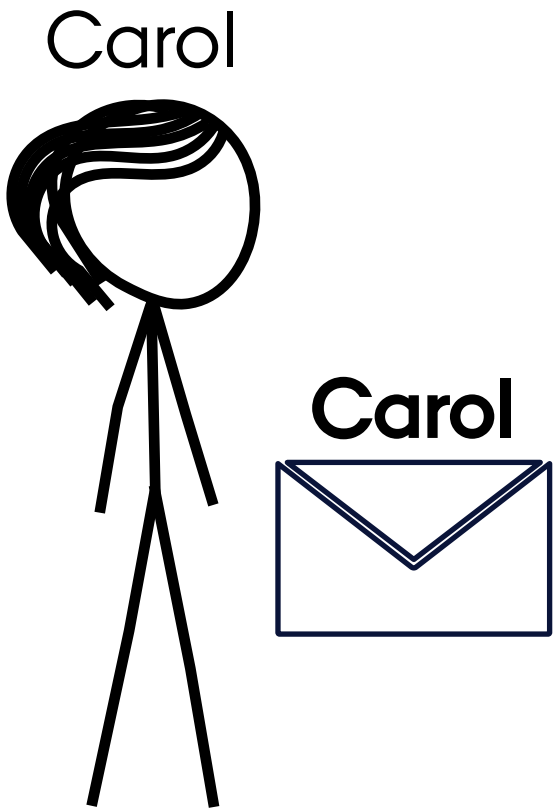
Carol

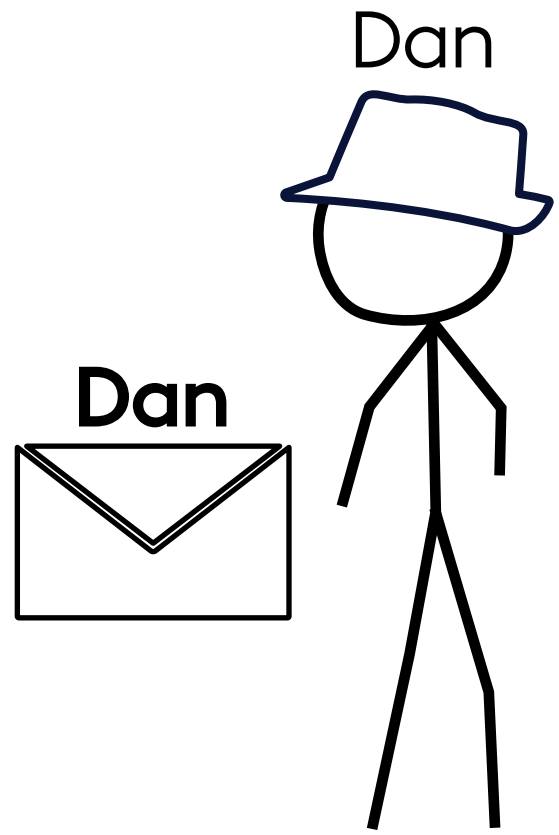


Dan



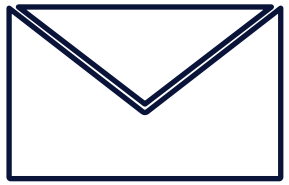
EXAMPLE: AUCTION



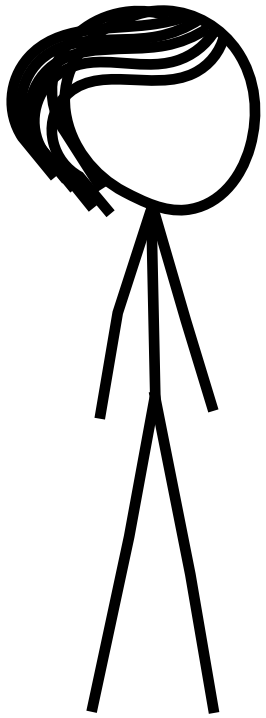


EXAMPLE: AUCTION

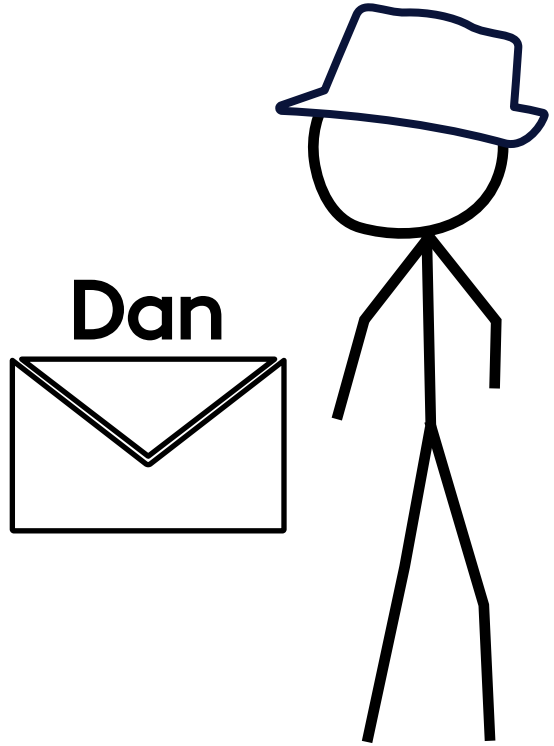
Carol



Carol

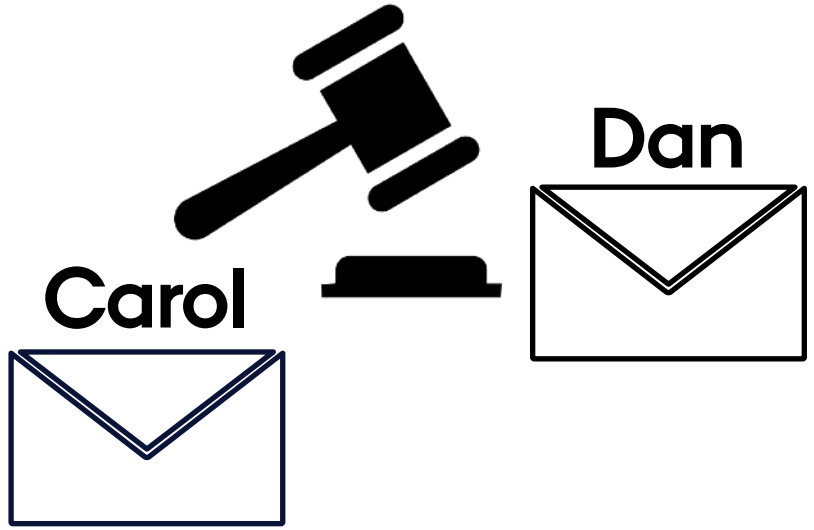


Dan

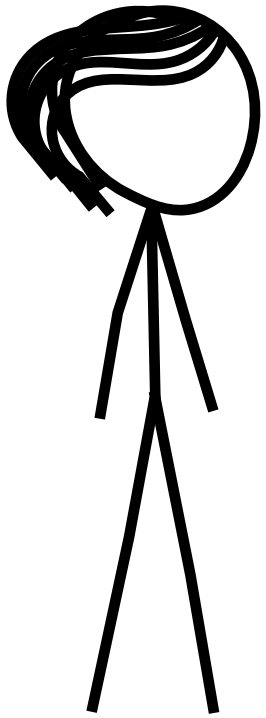


Dan

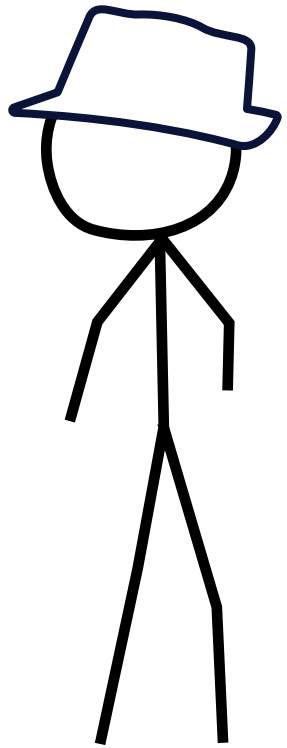
EXAMPLE: AUCTION



Carol



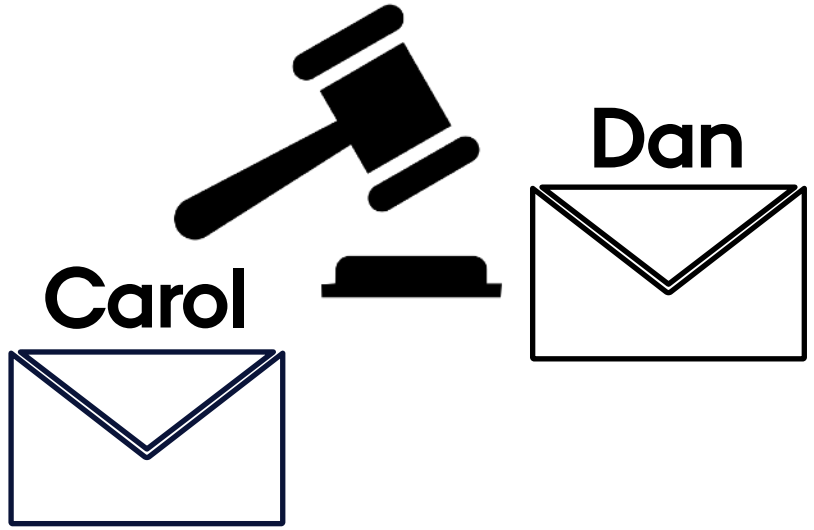
Dan



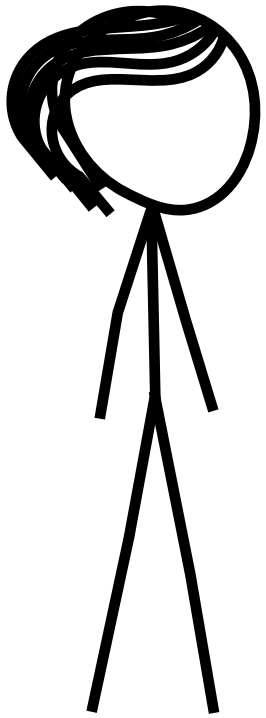
EXAMPLE: AUCTION

Carol $\not\subseteq$ **Dan**

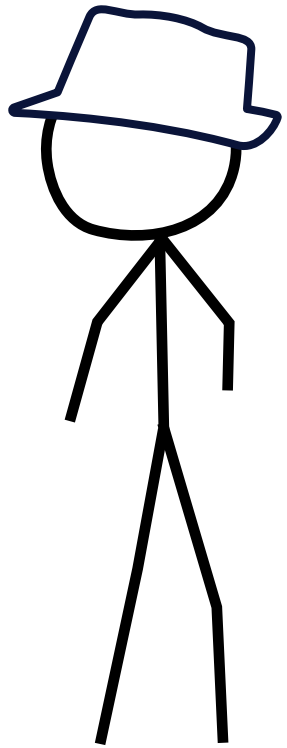
Dan $\not\subseteq$ **Carol**



Carol



Dan

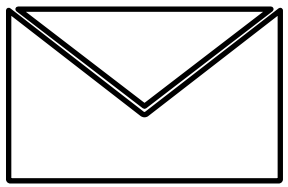


EXAMPLE: AUCTION

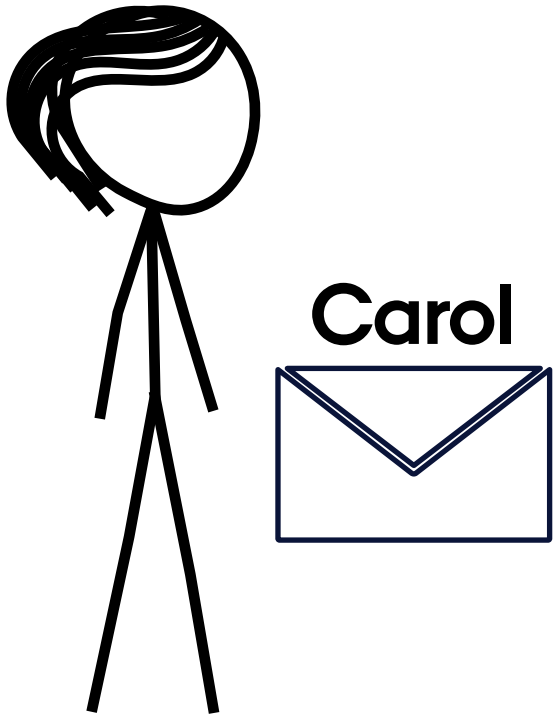
Carol $\not\subseteq$ **Dan**

Dan $\not\subseteq$ **Carol**

Dan

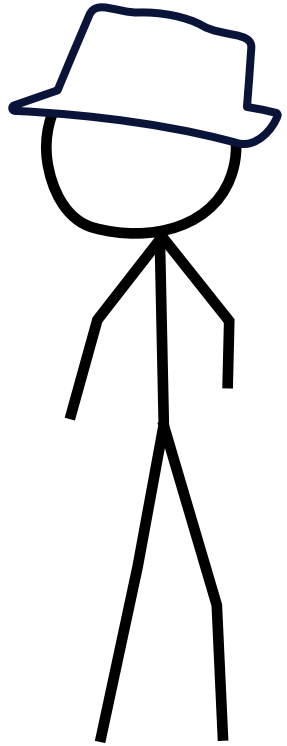


Carol



Carol

Dan

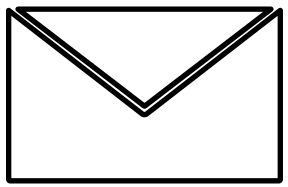


EXAMPLE: AUCTION

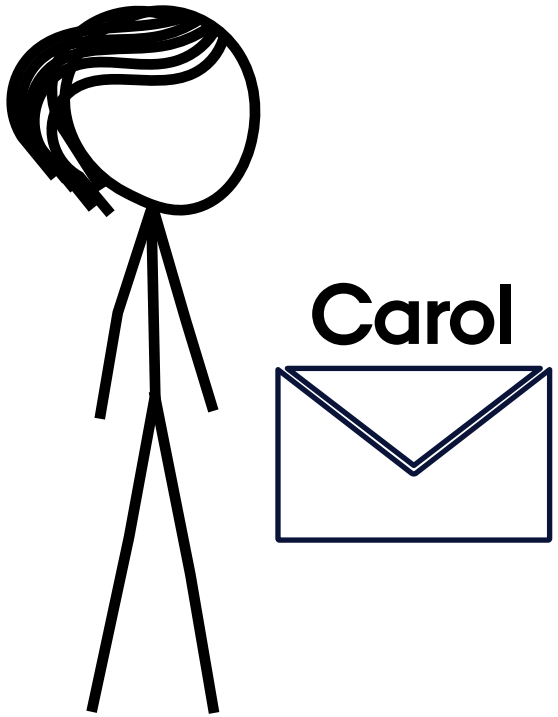
Carol $\not\subseteq$ **Dan**

Dan $\not\subseteq$ **Carol**

Dan

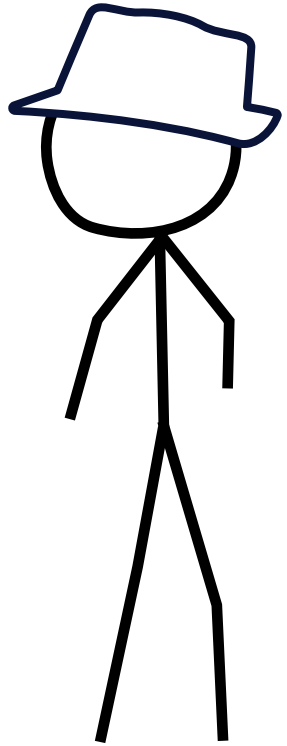


Carol



Carol

Dan

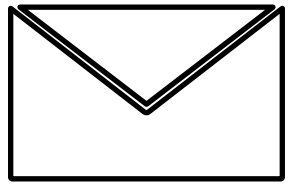


EXAMPLE: AUCTION

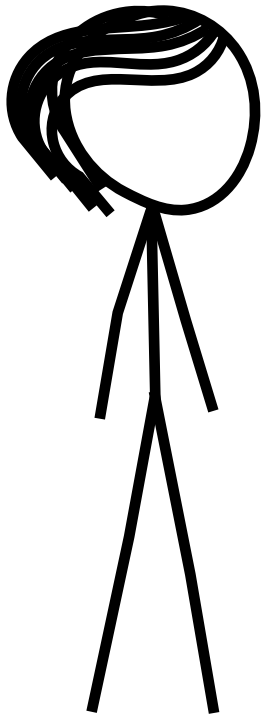
Carol $\not\supseteq$ **Dan**

Dan $\not\supseteq$ **Carol**

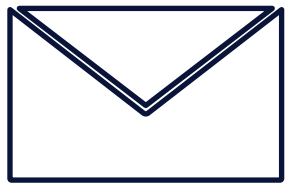
Dan



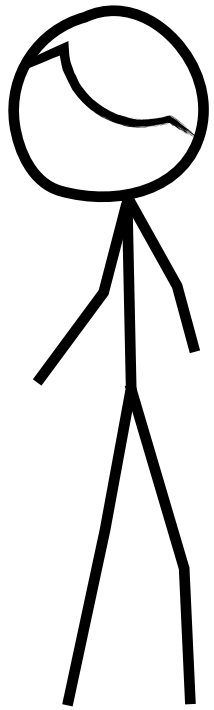
Carol



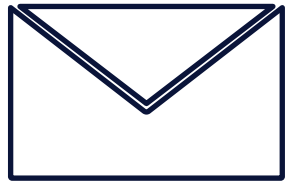
Carol



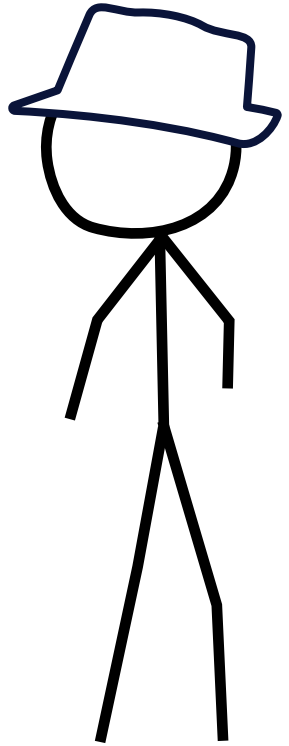
Erin



Erin



Dan



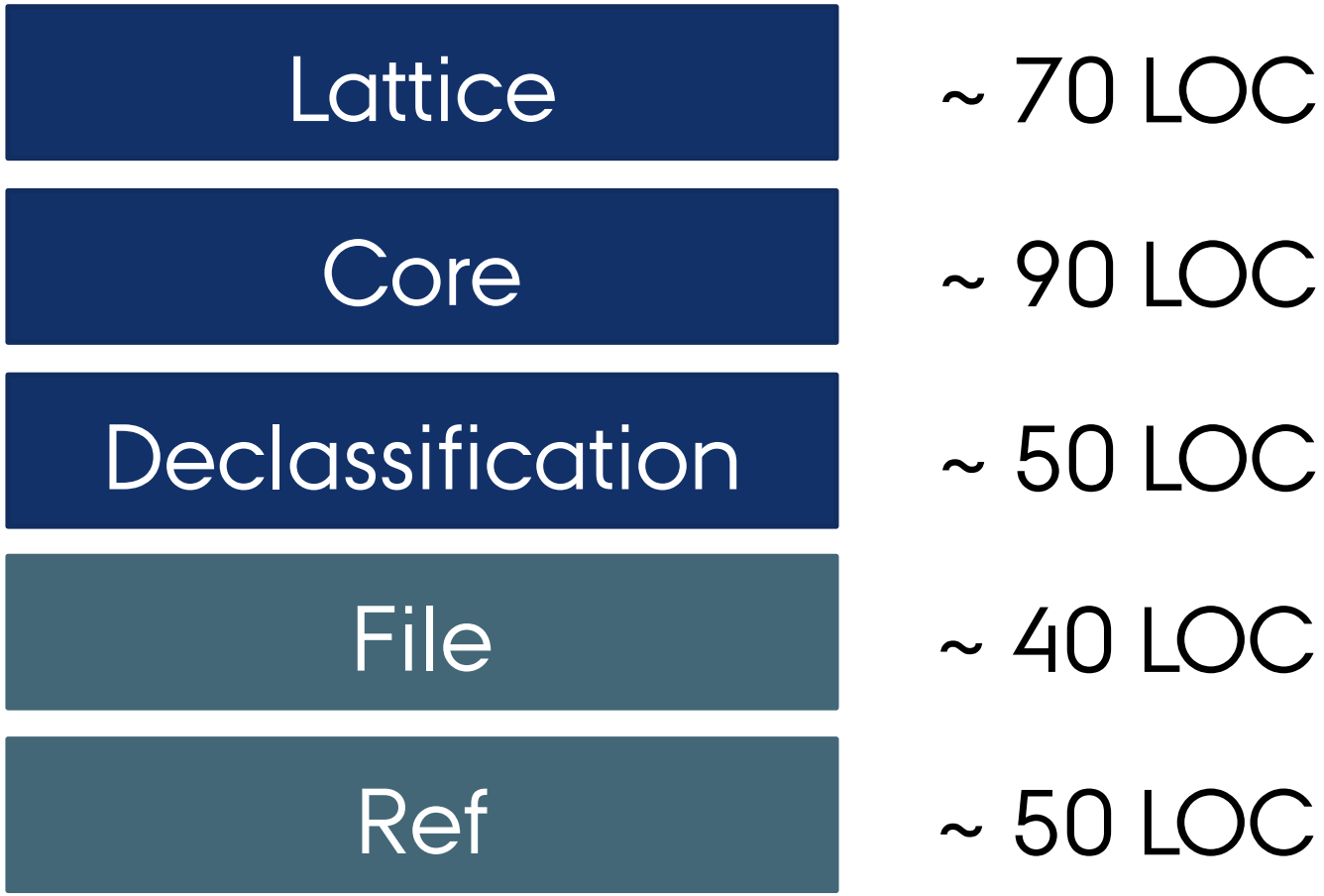
THIS WORK

- **DEPSEC**: a static information-flow control (IFC) library for Idris
- Showcases how dependent types increase the expressiveness of state-of-the-art IFC libraries
- Novel and powerful means for statically-enforced declassification using dependent types

Idris (Brady, 2013)

- **General-purpose**, pure functional language
- Features full dependent types
- Heavily inspired by Haskell

DepSec - Overview



DepSec - Overview

Lattice ~ 70 LOC

Core ~ 90 LOC

DepSec - Lattice

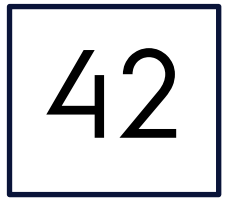
```
interface JoinSemilattice a where
  join        : a -> a -> a
  commutative : (x, y : a) -> x `join` y = y `join` x
  -- ...

interface Poset a where
  leq         : a -> a -> Type
  -- ...

implementation JoinSemilattice a => Poset a where
  leq x y = (x `join` y = y)
  -- ...

interface JoinSemilattice a => BoundedJoinSemilattice a where
  Bottom : a
  -- ...
```

DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```

DepSec - CORE

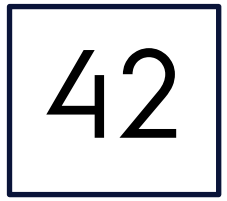
```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```



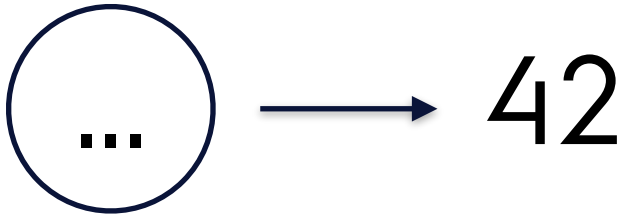

```
Labeled H Int
```

DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```



Labeled H Int

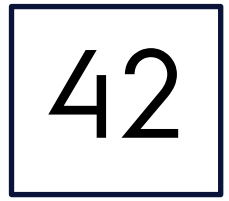


DIO H Int

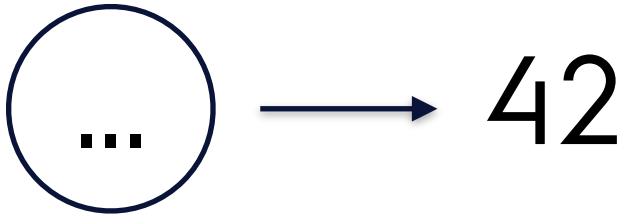
DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```

```
label : Poset labelType
     => {l : labelType}
     -> (value : a)
     -> Labeled l a
```



Labeled H Int

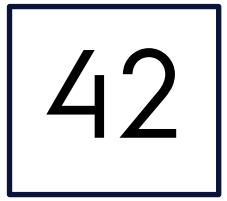


DIO H Int

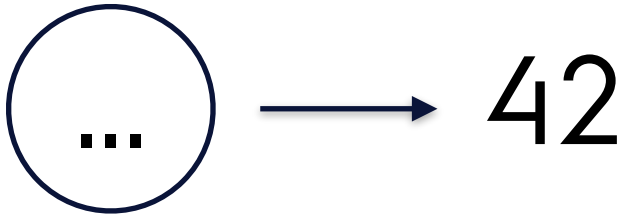
DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```

```
label : Poset labelType
     => {l : labelType}
     -> (value : a)
     -> Labeled l a

unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```



Labeled H Int



DIO H Int

DEPSEC - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```

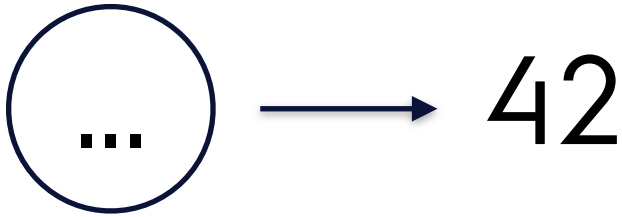
```
label : Poset labelType
     => {l : labelType}
     -> (value : a)
     -> Labeled l a

unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```

implicit parameter

context search

42

Labeled H Int

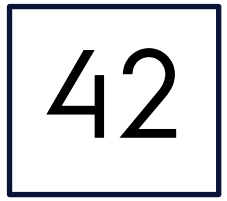


DIO H Int

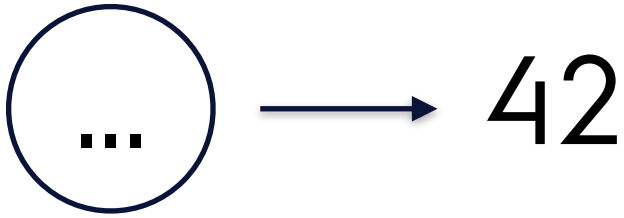
DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```

```
label : Poset labelType
     => {l : labelType}
     -> (value : a)
     -> Labeled l a

unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```



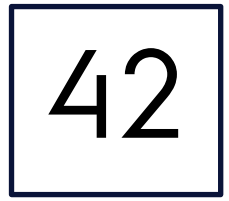
Labeled H Int



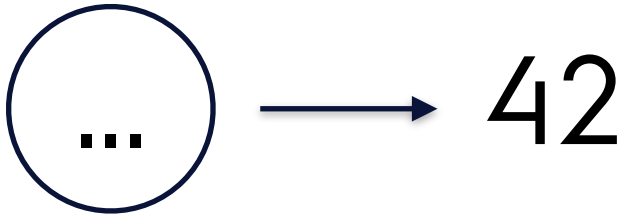
DIO H Int

DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```



Labeled H Int

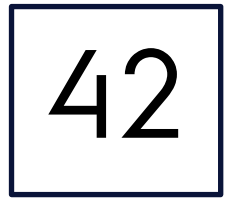


DIO H Int

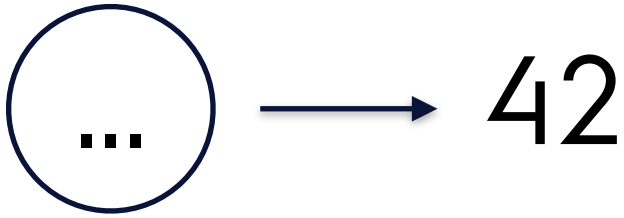
```
label : Poset labelType
     => {l : labelType}
     -> (value : a)
     -> Labeled l a

unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a

plug : Poset labelType
    => {l,l' : labelType}
    -> {auto flow : l `leq` l'}
    -> DIO l' a
    -> DIO l (Labeled l' a)
```

DepSec - CORE

```
data Labeled : l -> Type -> Type where
  MkLabeled : vt -> Labeled l vt -- TCB

data DIO : l -> Type -> Type where
  MkDIO : IO vt -> DIO l vt -- TCB

implementation Monad (DIO l) where
  -- ...
```

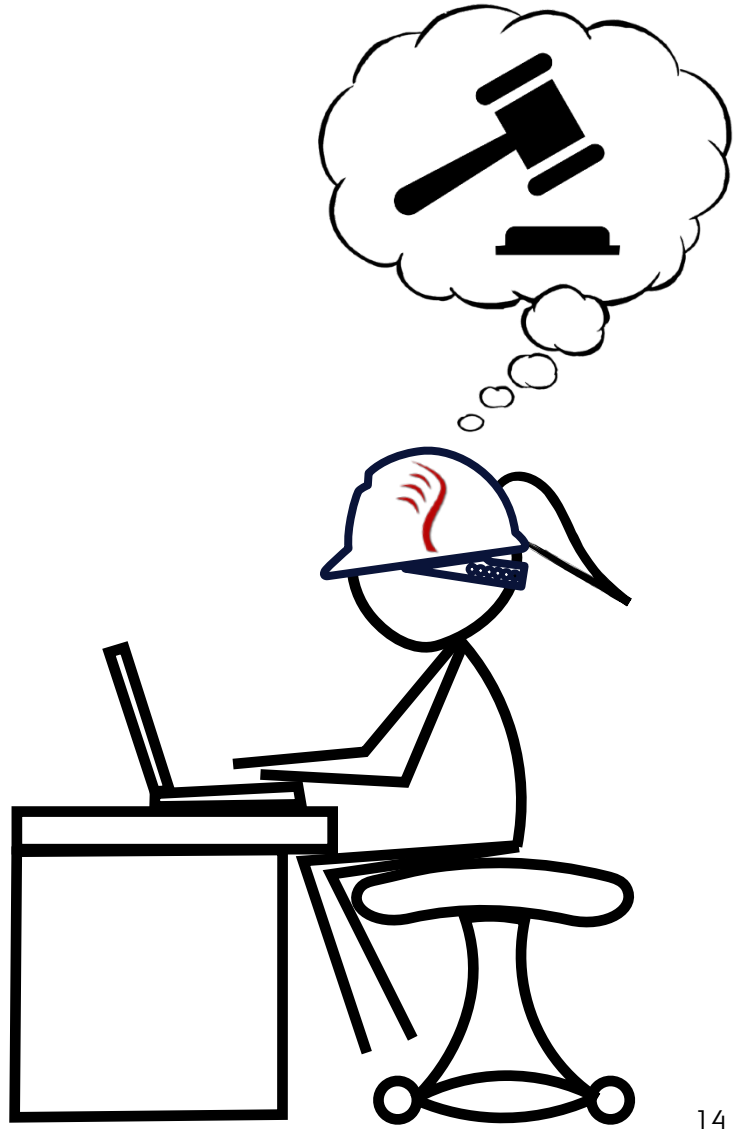


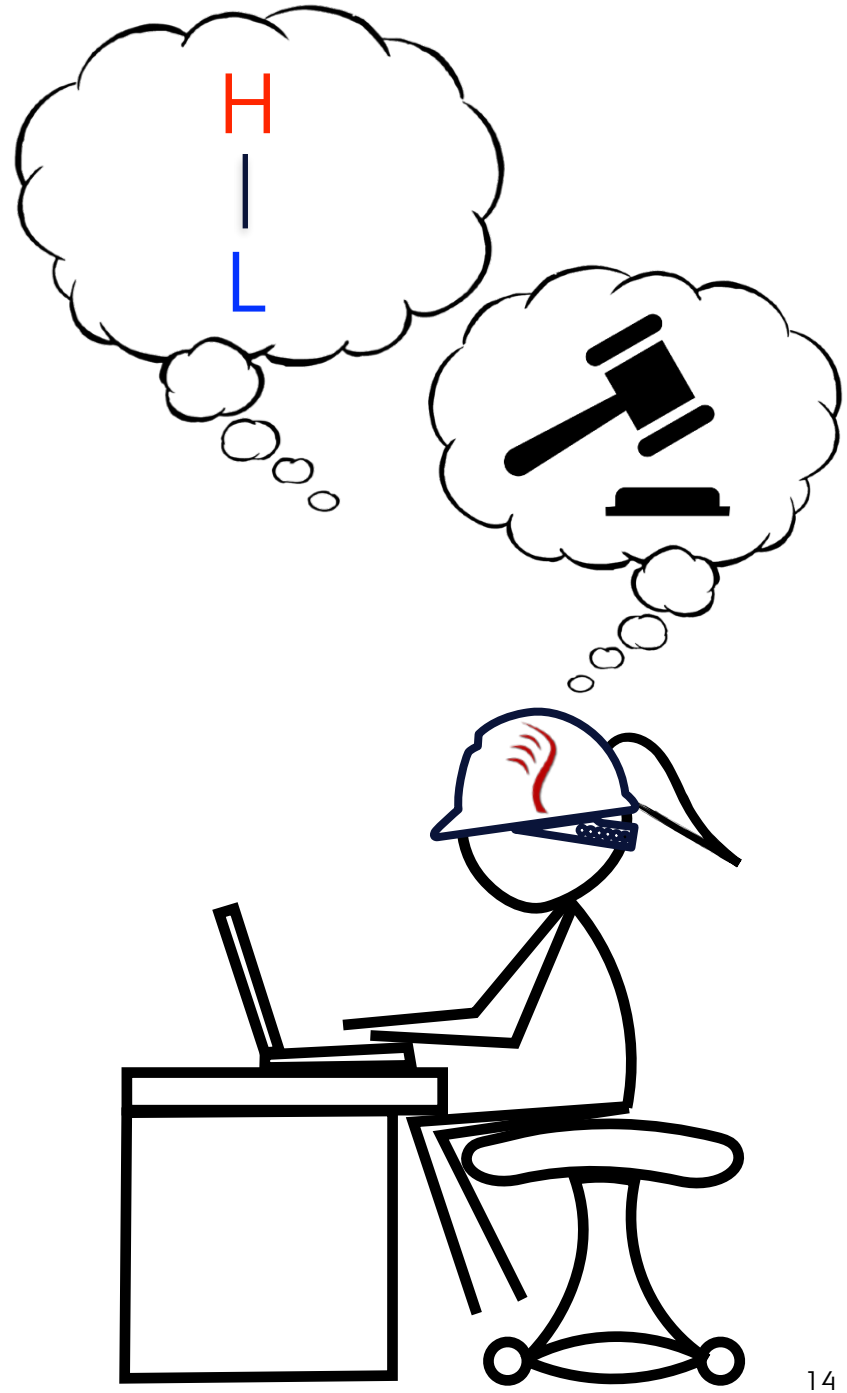
Labeled H Int



DIO H Int

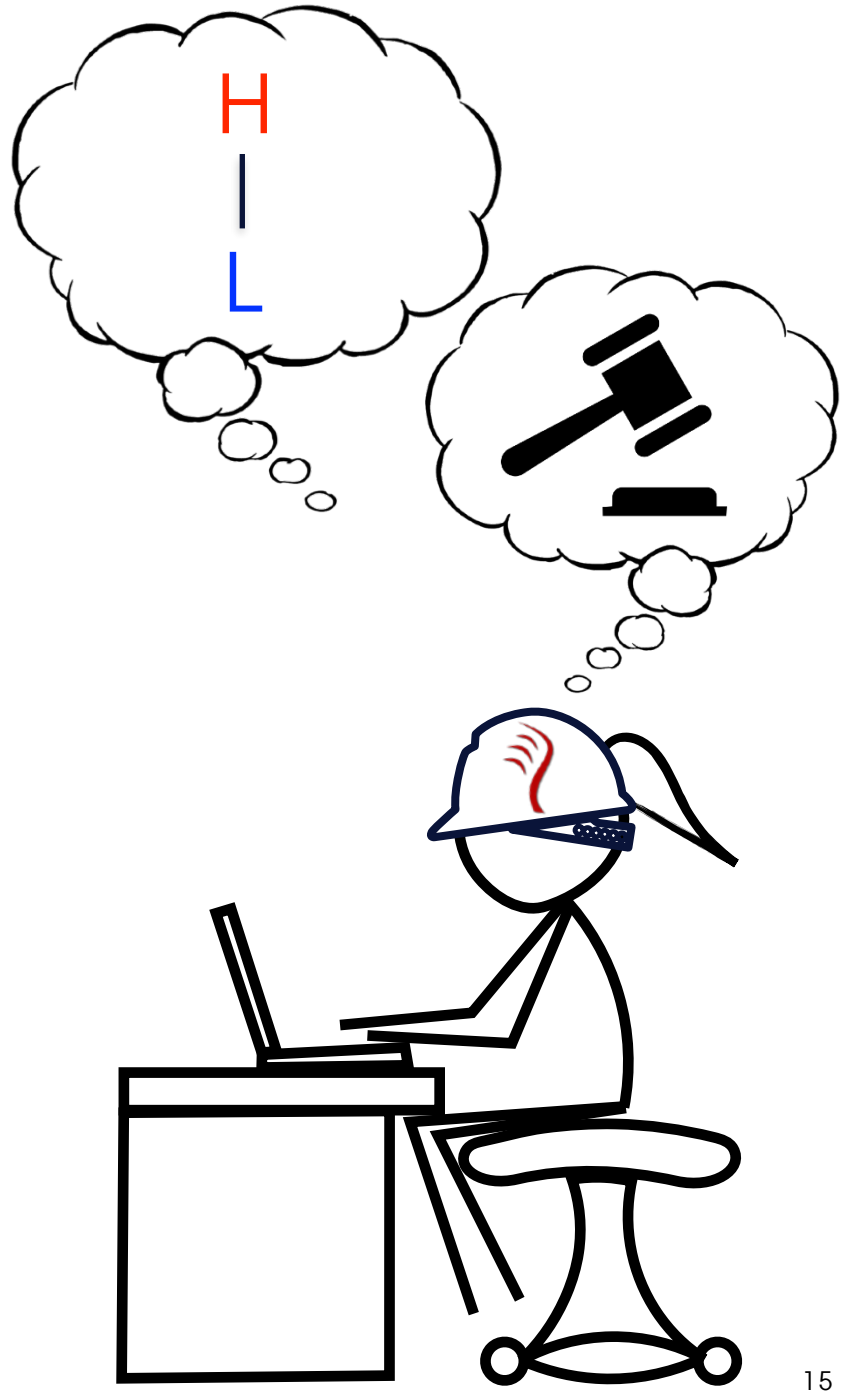
```
label : Poset labelType
     => {l : labelType}
     -> (value : a)
     -> Labeled l a

unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a

plug : Poset labelType
    => {l,l' : labelType}
    -> {auto flow : l `leq` l'}
    -> DIO l' a
    -> DIO l (Labeled l' a)

relabel : Poset labelType
       => {l, l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> Labeled l' a
```




H
I
L

AUCTION REQUIREMENTS

- Bids should be confidential
- Only winning bid should be disclosed
- Alice will put as little effort as possible into this project



AUCTION I - SETUP

AUCTION I - SETUP

```
data TwoPoint = H | L

implementation JoinSemilattice TwoPoint where
  -- ...
```

AUCTION I - SETUP

```
data TwoPoint = H | L

implementation JoinSemilattice TwoPoint where
  -- ...
```

```
User : Type
User = Nat

Bid : Type
Bid = (User, Nat)
```

AUCTION I - SETUP

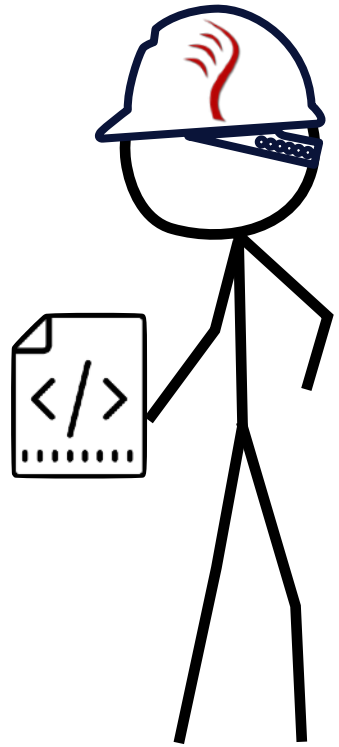
```
data TwoPoint = H | L

implementation JoinSemilattice TwoPoint where
  -- ...
```

```
User : Type
User = Nat

Bid : Type
Bid = (User, Nat)

BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

AUCTION I - SETUP

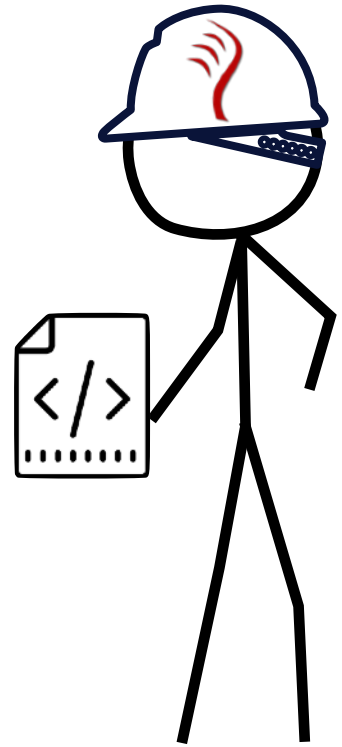
```
data TwoPoint = H | L

implementation JoinSemilattice TwoPoint where
  -- ...

User : Type
User = Nat

Bid : Type
Bid = (User, Nat)

BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

AUCTION I



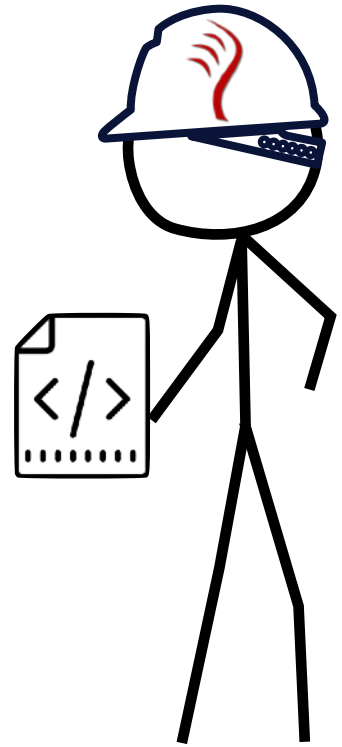
AUCTION I

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```



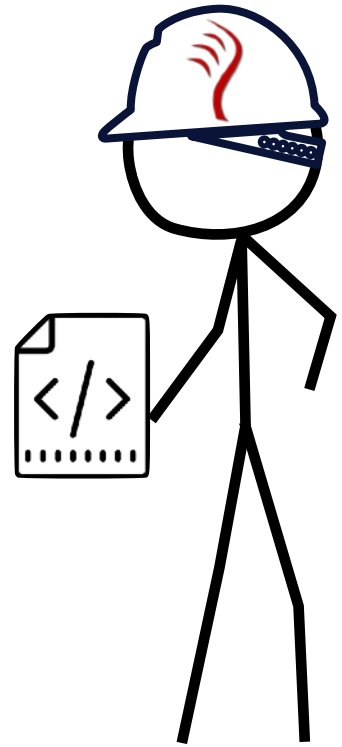
AUCTION I

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```



AUCTION I

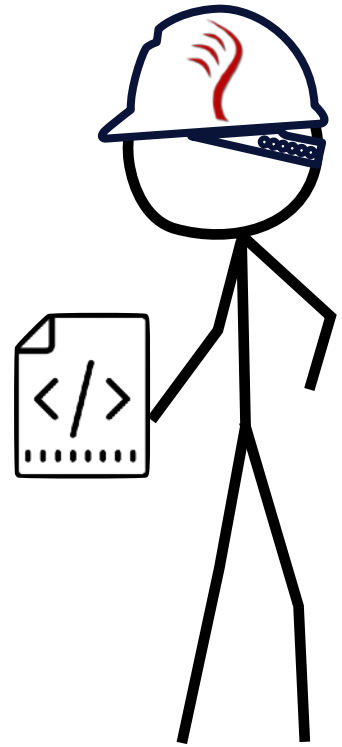
```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```



AUCTION I

```
unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```

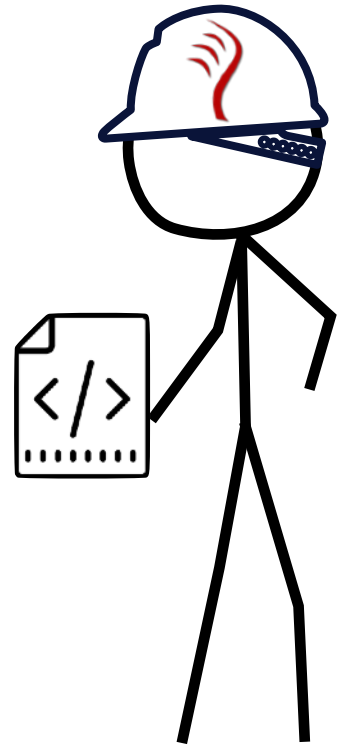


AUCTION I

```
BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

```
unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```

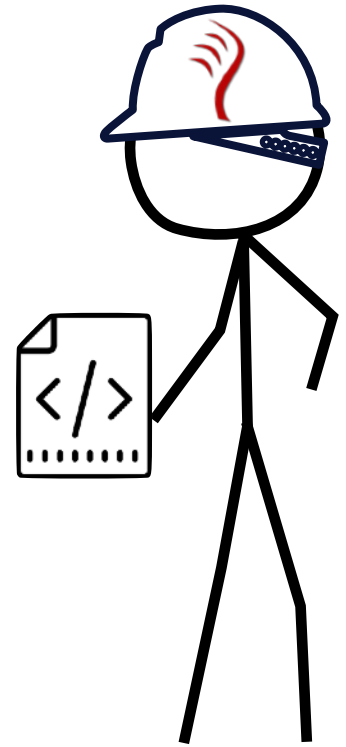


AUCTION I

```
BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

```
unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```


```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```

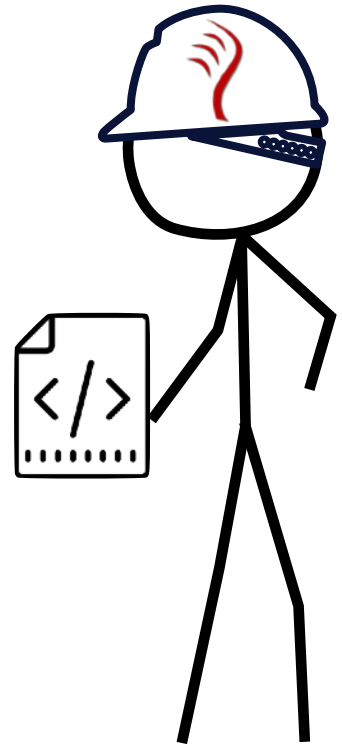


AUCTION I

```
BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

```
unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```



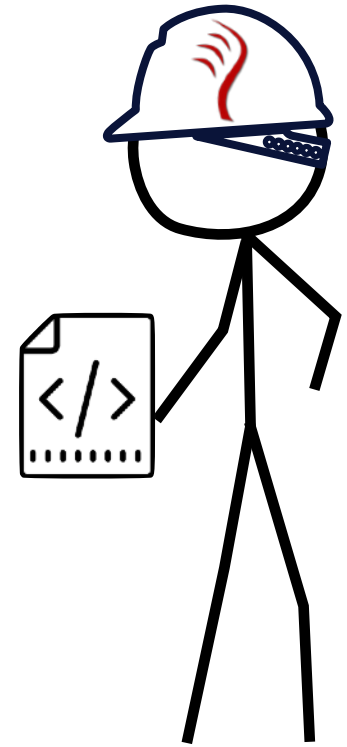
AUCTION I

```
BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

```
unlabel : Poset labelType
       => {l,l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> DIO l' a
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'
```

return

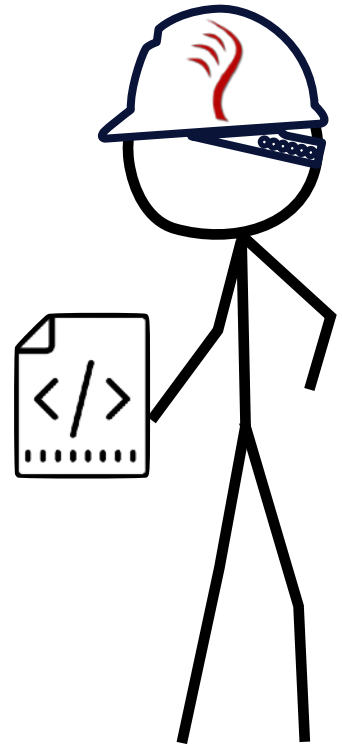


AUCTION I

```
BidLog : Nat -> Type
BidLog n = Vect n (Labeled H Bid)
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'

getWinner : BidLog (S n) -> DIO L (Labeled H Bid)
getWinner log = plug $ computeMaxBid log
```

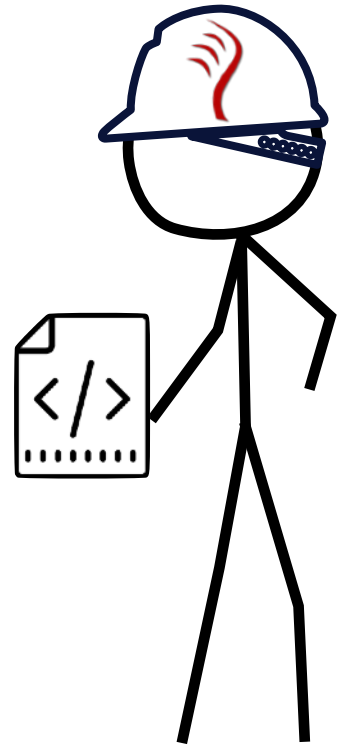
return



AUCTION I

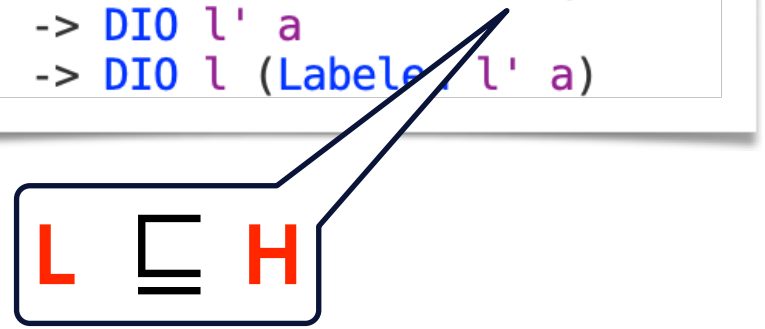
```
plug : Poset labelType
    => {l,l' : labelType}
    -> {auto flow : l `leq` l'}
    -> DIO l' a
    -> DIO l (Labeled l' a)
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'

getWinner : BidLog (S n) -> DIO L (Labeled H Bid)
getWinner log = plug $ computeMaxBid log
```

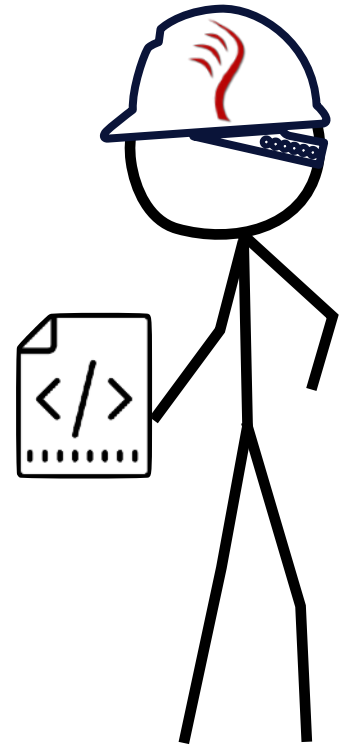


AUCTION I

```
plug : Poset labelType
    => {l,l' : labelType}
    -> {auto flow : l `leq` l'}
    -> DIO l' a
    -> DIO l (Labeled l' a)
```



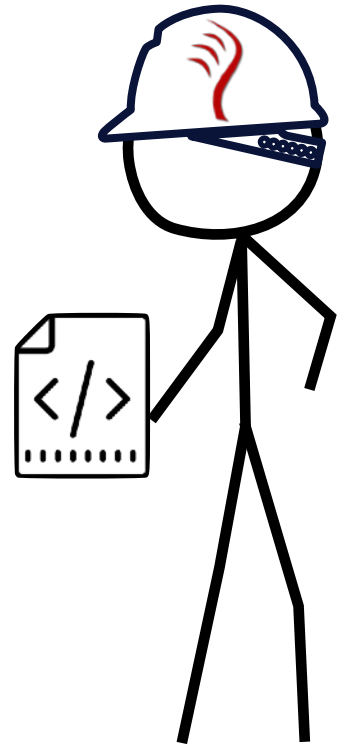

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'

getWinner : BidLog (S n) -> DIO L (Labeled H Bid)
getWinner log = plug $ computeMaxBid log
```



AUCTION I

```
plug : Poset labelType
    => {l,l' : labelType}
    -> {auto flow : l `leq` l'}
    -> DIO l' a
    -> DIO l (Labeled l' a)
```

```
maxBid : Bid -> Bid -> Bid
-- ...

computeMaxBid : BidLog (S n) -> DIO H Bid
computeMaxBid (x :: []) = unlabel x
computeMaxBid (x :: tail@(y :: xs)) =
  do x' <- unlabel x
     tail' <- computeMaxBid tail
     pure $ maxBid x' tail'

getWinner : BidLog (S n) -> DIO L (Labeled H Bid)
getWinner log = plug $ computeMaxBid log
```



AUCTION I

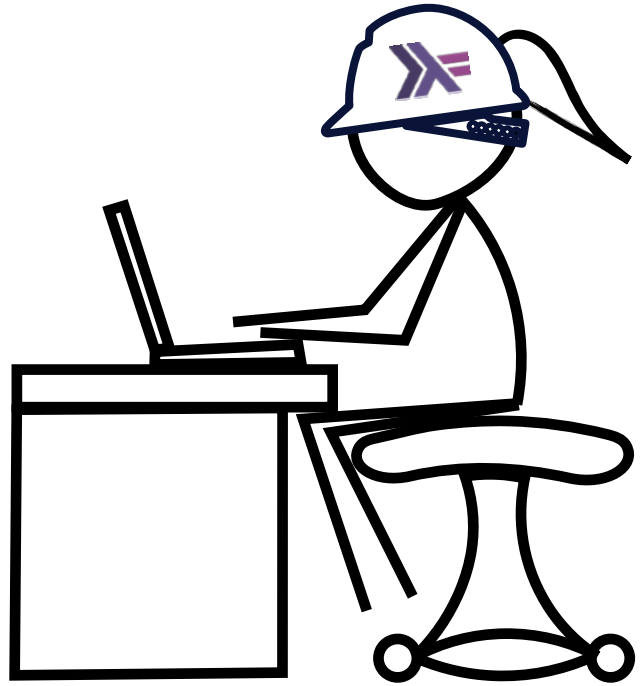
```
main : IO ()
main =
  do putStrLn "##### Welcome to the auction! #####"
     putStrLn "Taking bids:"
     let log : BidLog _ = [!getBid, !getBid]
     putStrLn "Announcing winner:"
     winner <- run $ getWinner log
     let (MkLabeled x) = winner
     putStrLn $ show x
     putStrLn "##### Bye bye! #####"
```

AUCTION I

```
main : IO ()
main =
  do putStrLn "##### Welcome to the auction! #####"
     putStrLn "Taking bids:"
     let log : BidLog _ = [!getBid, !getBid]
     putStrLn "Announcing winner:"
     winner <- run $ getWinner log
     let (MkLabeled x) = winner
     putStrLn $ show x
     putStrLn "##### Bye bye! #####"
```

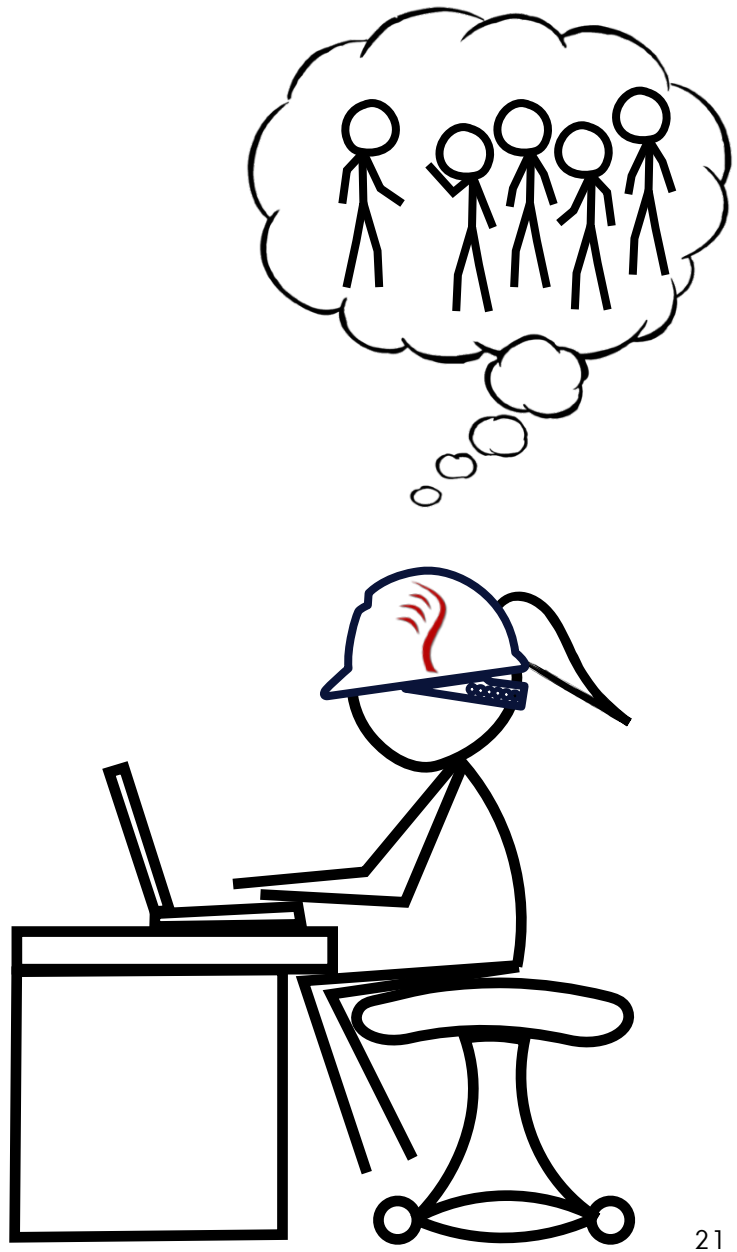
AUCTION I

```
main : IO ()
main =
  do putStrLn "##### Welcome to the auction! #####"
     putStrLn "Taking bids:"
     let log : BidLog _ = [!getBid, !getBid]
     putStrLn "Announcing winner:"
     winner <- run $ getWinner log
     let (MkLabeled x) = winner
     putStrLn $ show x
     putStrLn "##### Bye bye! #####"
```

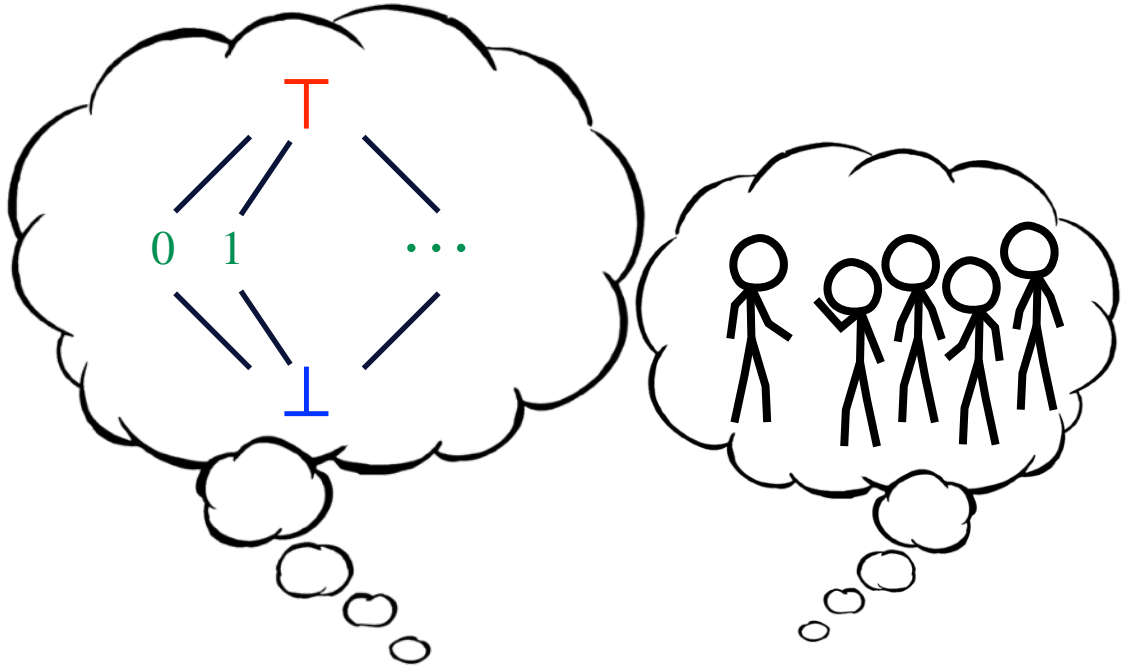


AUCTION II - SETUP

AUCTION II - SETUP

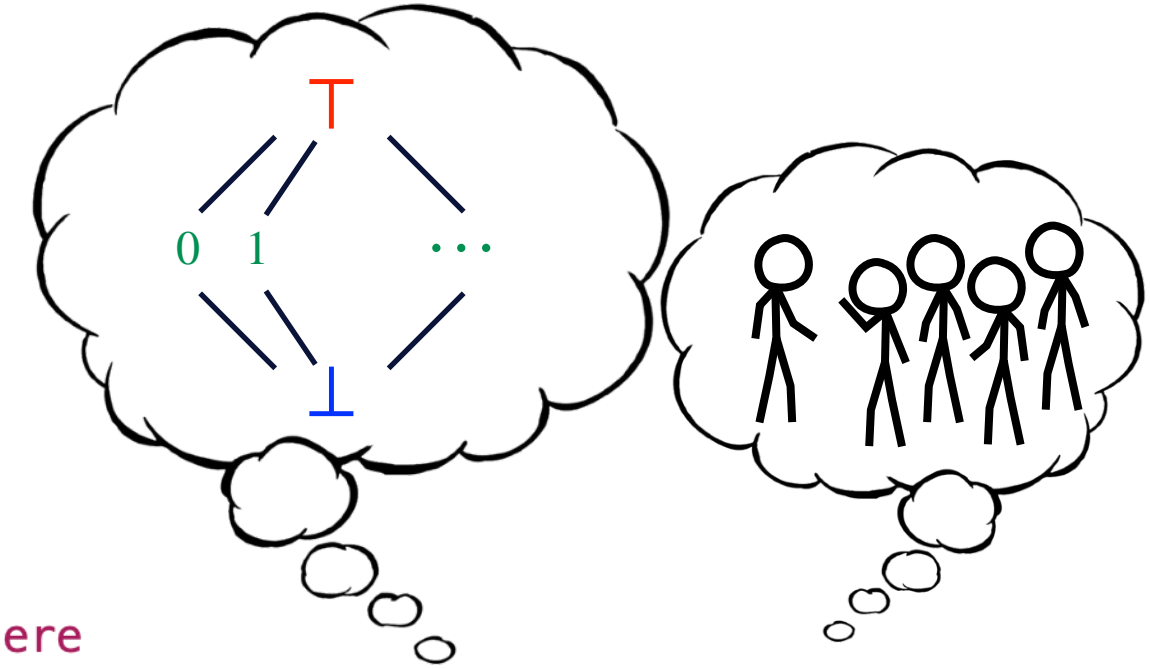


AUCTION II - SETUP



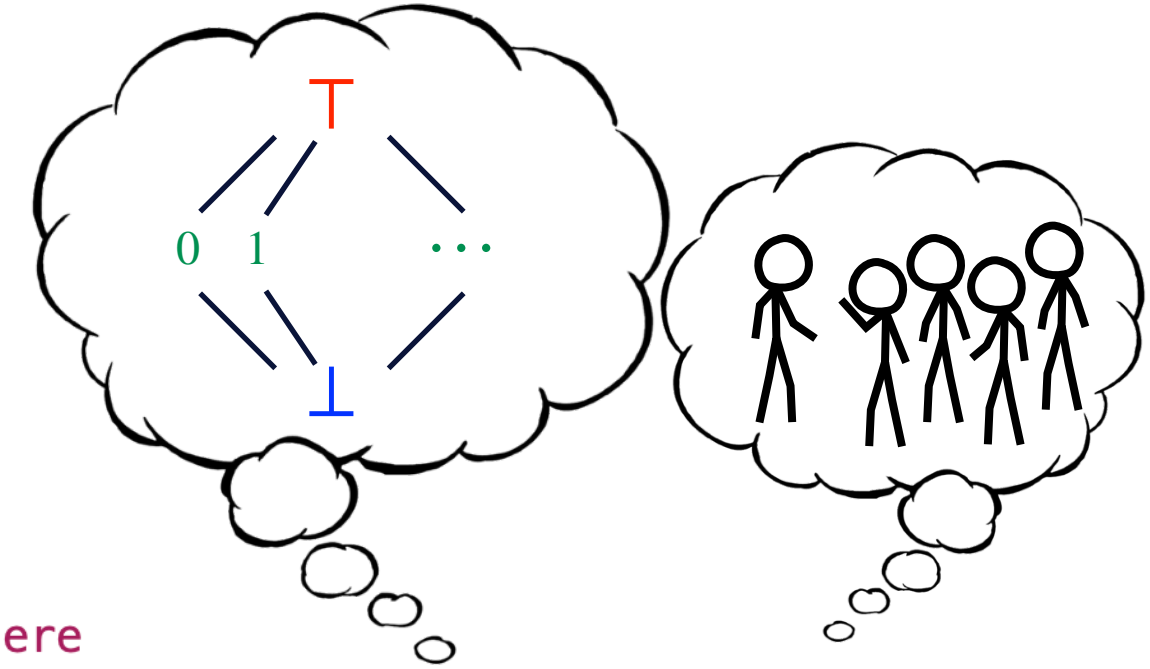
AUCTION II - SETUP

```
data Label : Type where
  Top  : Label
  U    : Nat -> Label
  Bot  : Label

implementation JoinSemilattice Label where
  -- ...

implementation BoundedJoinSemilattice Label where
  -- ...
```



AUCTION II - SETUP

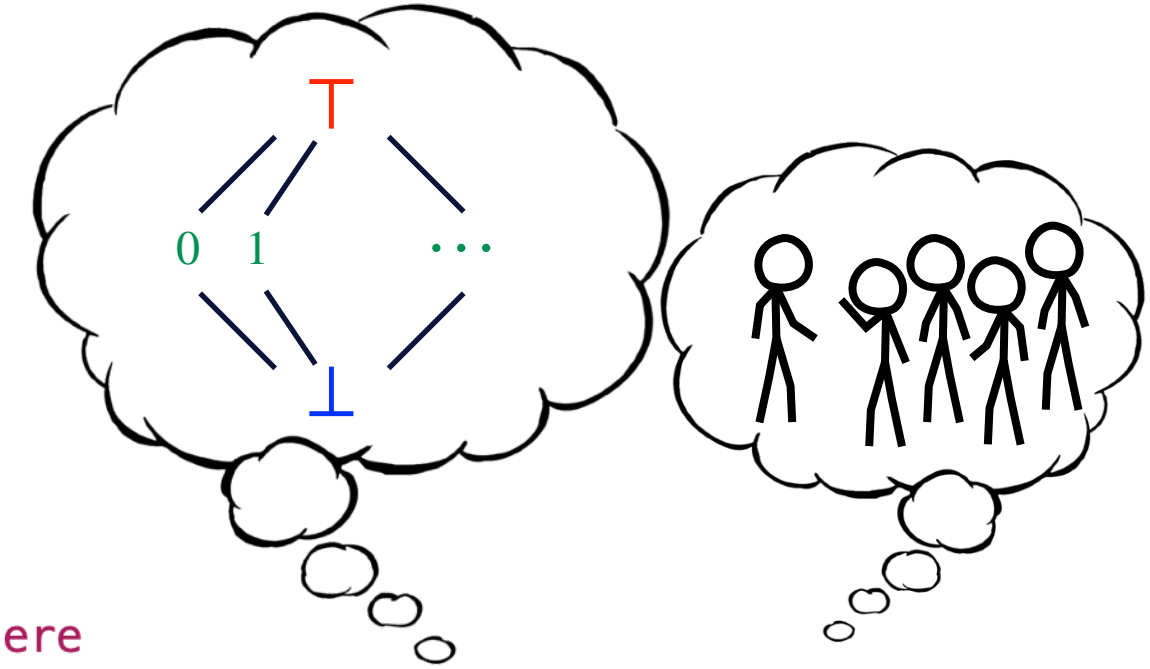
```
data Label : Type where
  Top  : Label
  U    : Nat -> Label
  Bot  : Label

implementation JoinSemilattice Label where
  -- ...

implementation BoundedJoinSemilattice Label where
  -- ...

LabeledBid : Type
LabeledBid = (l : Label ** Labeled l Bid)

BidLog : Nat -> Type
BidLog n = Vect n LabeledBid
```



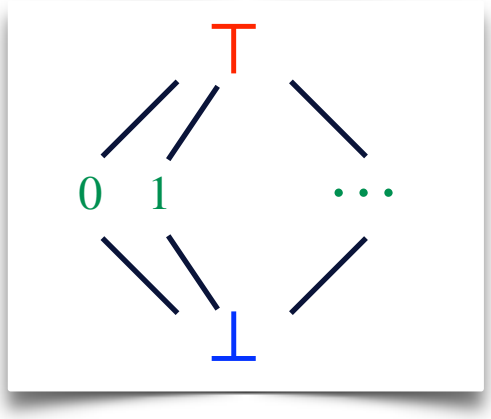
AUCTION II - SETUP

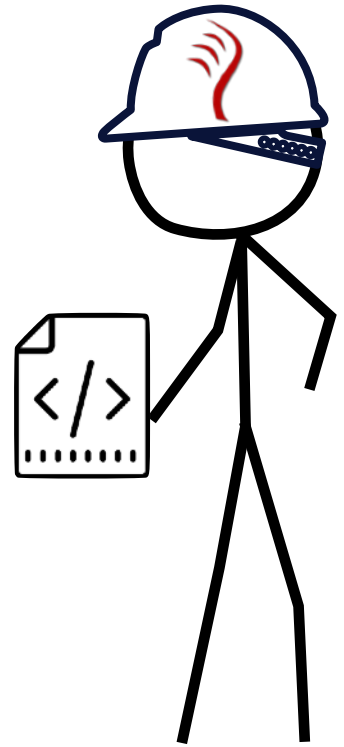
```
data Label : Type where
  Top  : Label
  U    : Nat -> Label
  Bot  : Label

implementation JoinSemilattice Label where
  -- ...

implementation BoundedJoinSemilattice Label where
  -- ...

LabeledBid : Type
LabeledBid = (l : Label ** Labeled l Bid)

BidLog : Nat -> Type
BidLog n = Vect n LabeledBid
```

$\exists l$: Label . Labeled l Bid



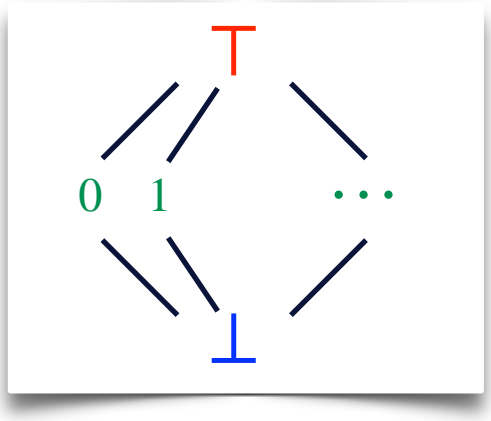
AUCTION II

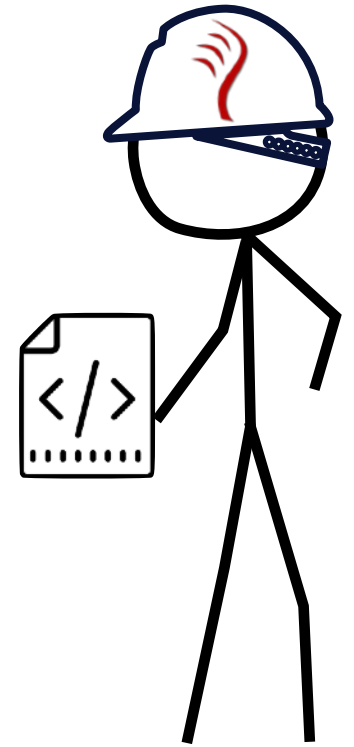




AUCTION II

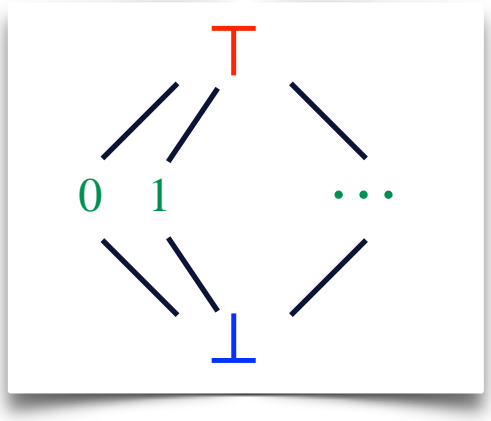
```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```

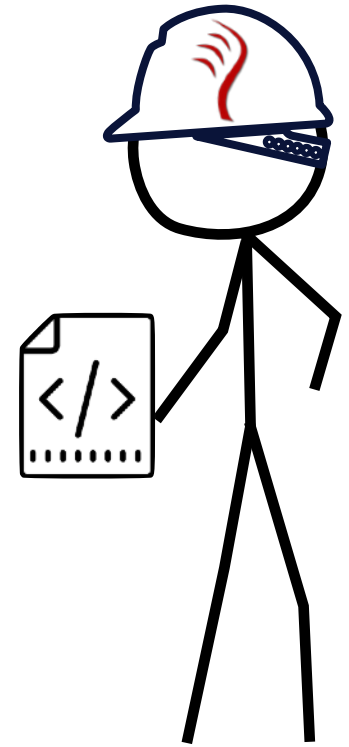




AUCTION II

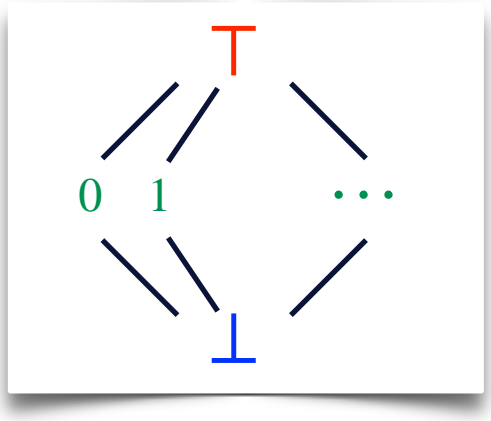
```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```

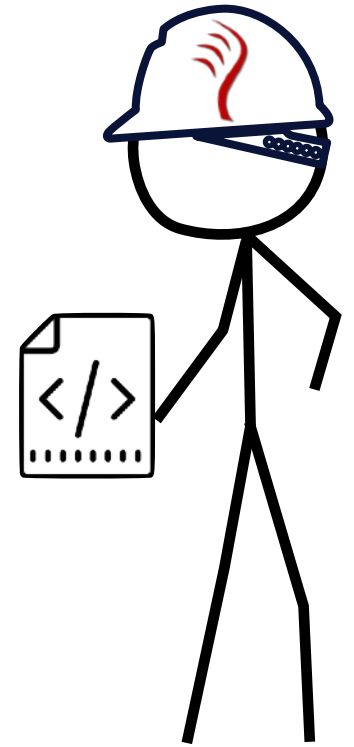




AUCTION II

```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```

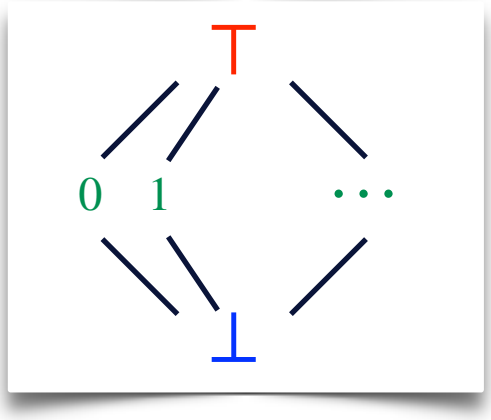


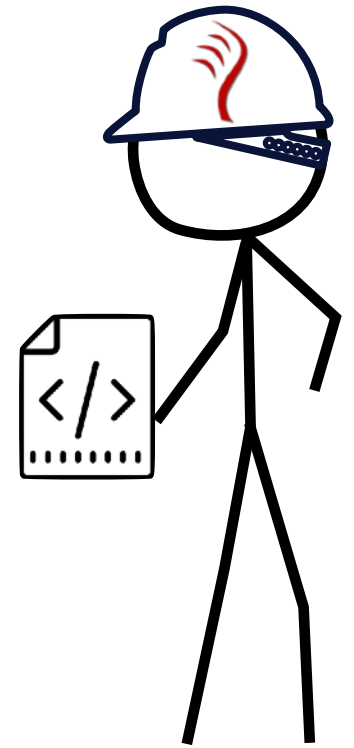


AUCTION II

```
relabel : Poset labelType
       => {l, l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> Labeled l' a
```

```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```





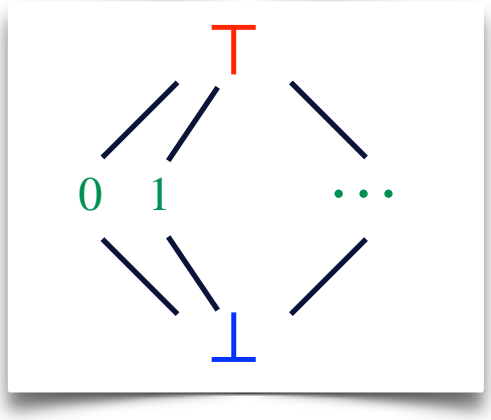
AUCTION II

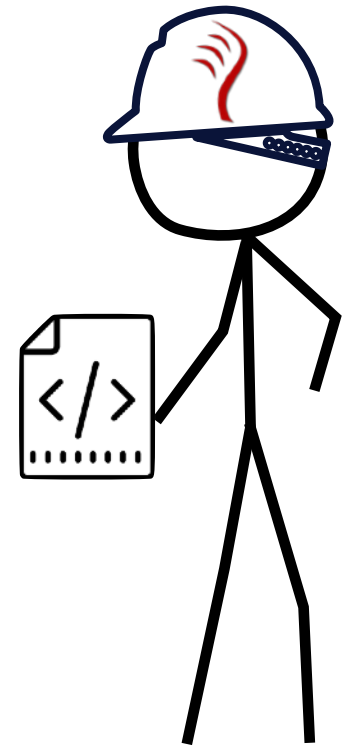
```
leq_l_Top : l `leq` Top
```

```
LabeledBid : Type
LabeledBid = (l : Label ** Labeled l Bid)
```

```
relabel : Poset labelType
       => {l, l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> Labeled l' a
```

```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```





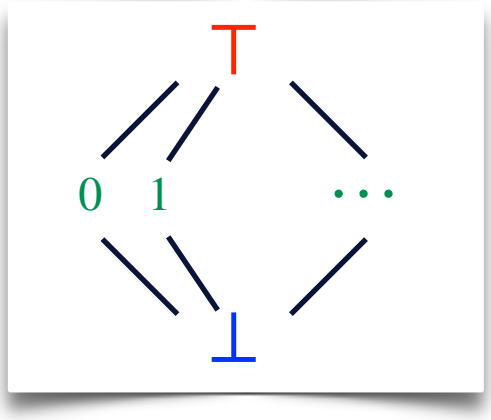
AUCTION II

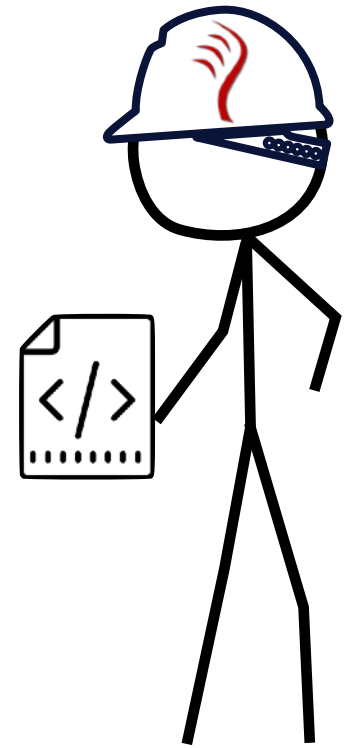
```
leq_l_Top : l `leq` Top
```

```
LabeledBid : Type
LabeledBid = (l : Label ** Labeled l Bid)
```

```
relabel : Poset labelType
       => {l, l' : labelType}
       -> {auto flow : l `leq` l'}
       -> Labeled l a
       -> Labeled l' a
```

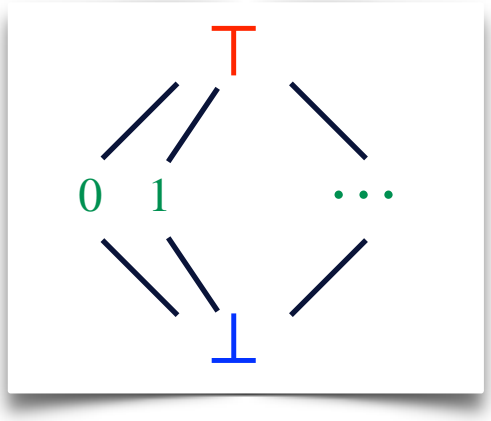
```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```

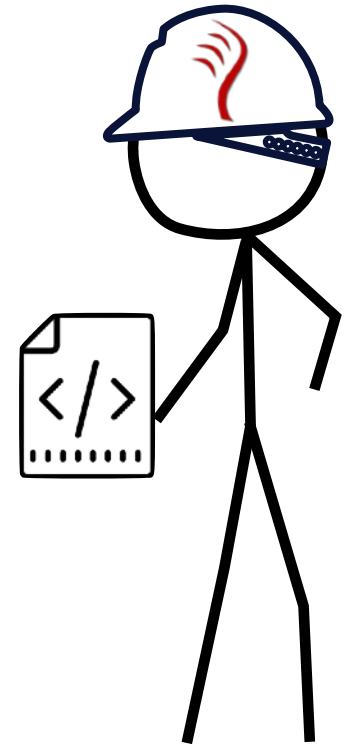




AUCTION II

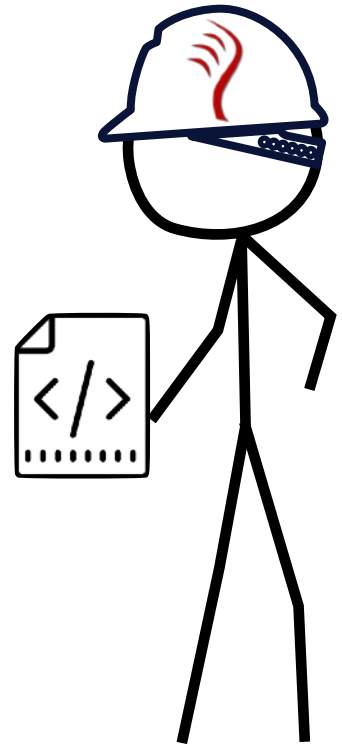
```
getWinner : (log : BidLog (S n)) -> DIO Bot (Labeled Top Bid)
getWinner ((l ** bid) :: []) = pure $ relabel {flow=leq_l_Top} bid
getWinner ((l ** bid) :: tail@(y :: ys)) =
  do let bid'  = relabel {flow=leq_l_Top} bid
     let tail' = !(getWinner tail)
     plug $ maxLabeledBid bid' tail'
  where
    maxLabeledBid : {a : Label}
                 -> Labeled a Bid
                 -> Labeled a Bid
                 -> DIO a Bid
```





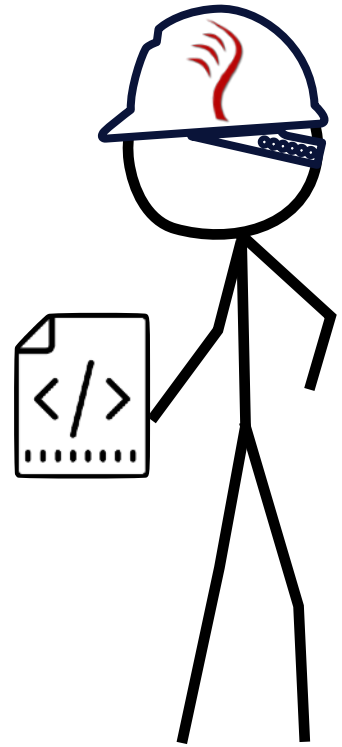
AUCTION II CONT'D

```
joinLabels : BoundedJoinSemilattice lt
          => Vect n (l : lt ** Labeled l a)
          -> lt
joinLabels [] = Bottom
joinLabels ((l ** _) :: xs) = join l $ joinLabels xs
```



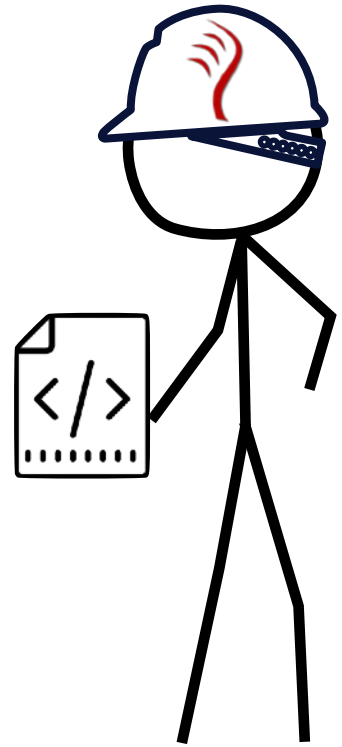
AUCTION II CONT'D

```
joinLabels : BoundedJoinSemilattice lt
          => Vect n (l : lt ** Labeled l a)
          -> lt
joinLabels [] = Bottom
joinLabels ((l ** _) :: xs) = join l $ joinLabels xs



getWinner' : (log : BidLog (S n))
          -> DIO Bot (Labeled (joinLabels log) Bid)
-- ...
```



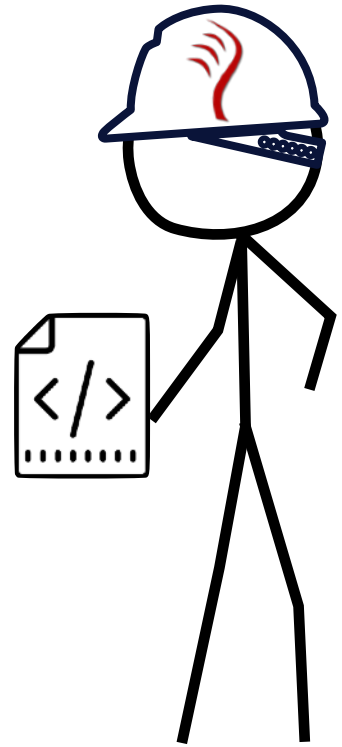
AUCTION II CONT'D

```
getWinner'' : BoundedJoinSemilattice a
           => (log : Vect (S n) (l : a ** Labeled l Bid))
           -> DIO (Bottom {a}) (Labeled (joinLabels log) Bid)
-- ...
  where
    maxLabeledBid : Poset a
                 => {z : a}
                 -> Labeled z Bid
                 -> Labeled z Bid
                 -> DIO z Bid
```



AUCTION II CONT'D

```
getWinner'' : BoundedJoinSemilattice a
           => (log : Vect (S n) (l : a ** Labeled l Bid))
           -> DIO (Bottom {a}) (Labeled (joinLabels log) Bid)
-- ...
  where
    maxLabeledBid : Poset a
                 => {z : a}
                 -> Labeled z Bid
                 -> Labeled z Bid
                 -> DIO z Bid
```



AUCTION II CONT'D

```
main : IO ()
main =
  do putStrLn "##### Welcome to the auction! #####"
     putStrLn "Taking bids:"
     let log : BidLog _ = [!getBid, !getBid]
     putStrLn "Announcing winner:"
     winner <- run $ getWinner'' log
     let (MkLabeled x) = winner
     putStrLn $ show x
     putStrLn "##### Bye bye! #####"
```

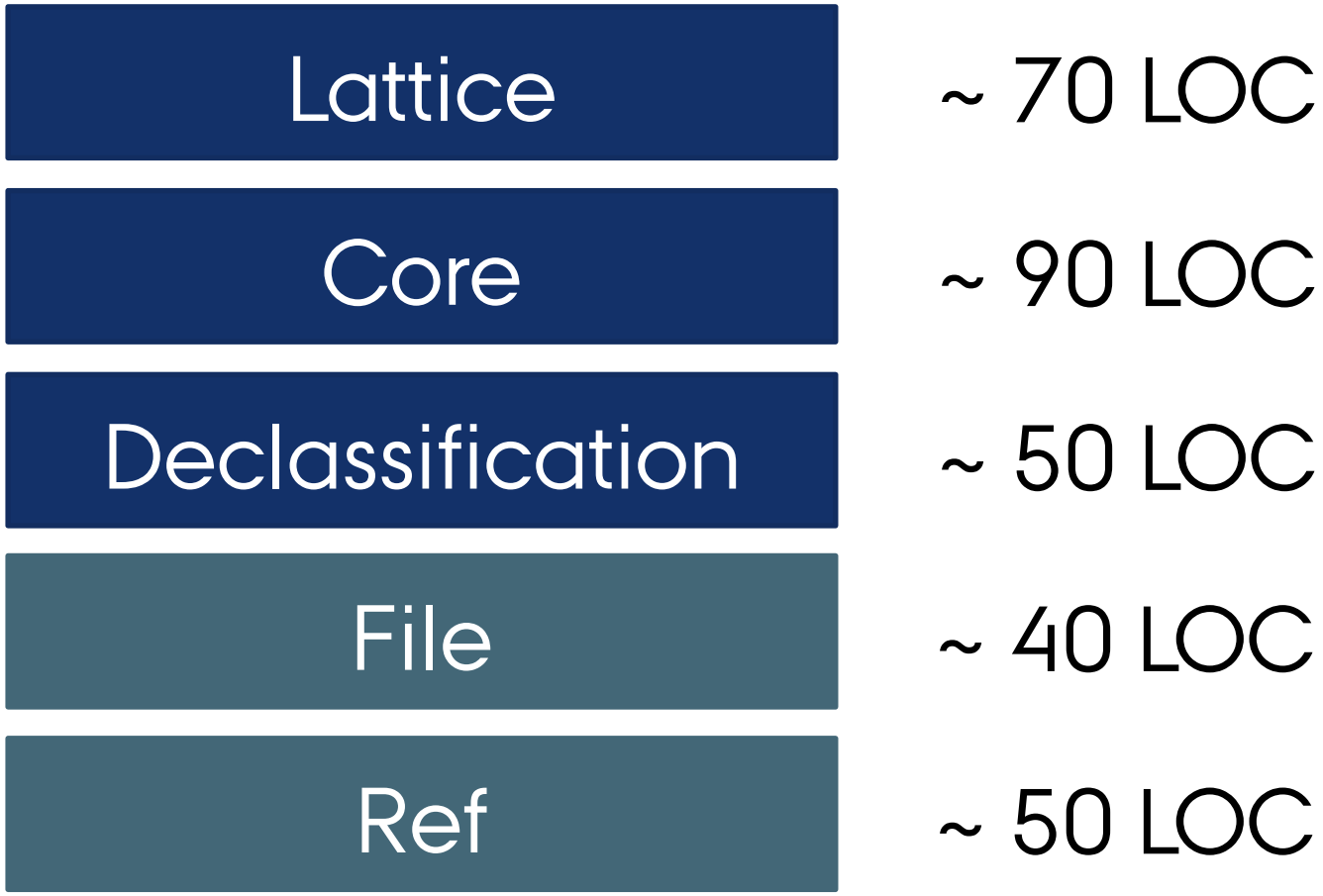
AUCTION II CONT'D

```
main : IO ()
main =
  do putStrLn "##### Welcome to the auction! #####"
     putStrLn "Taking bids:"
     let log : BidLog _ = [!getBid, !getBid]
     putStrLn "Announcing winner:"
     winner <- run $ getWinner'' log
     let (MkLabeled x) = winner
     putStrLn $ show x
     putStrLn "##### Bye bye! #####"
```

DECLASSIFICATION

- Can’t give full declassification powers to Bob
- Only the winning bid should be declassified
- How to address **what** is declassified?

DepSec - Overview

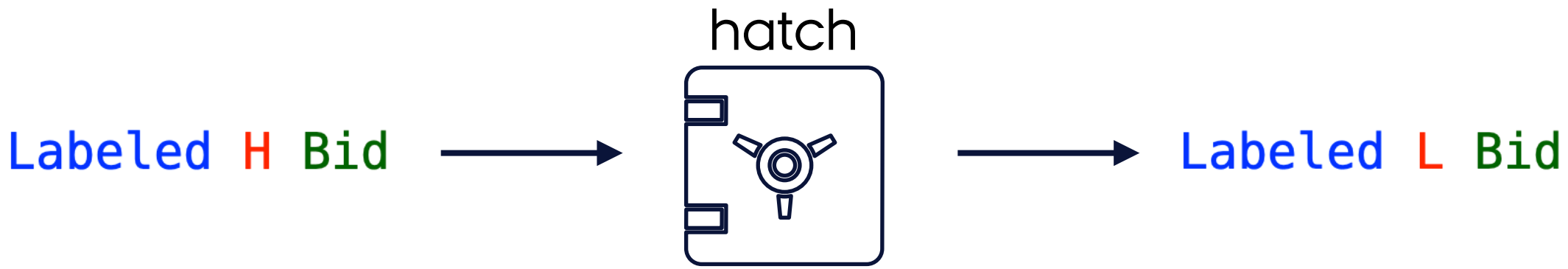


DEPSEC - OVERVIEW

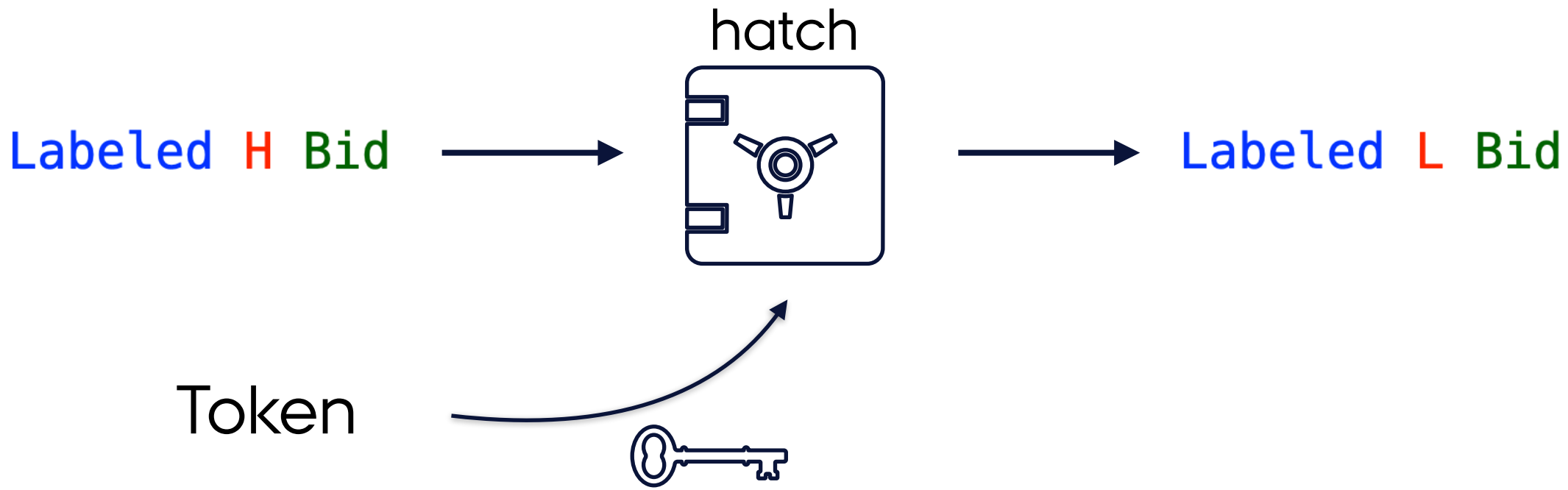
Declassification

~ 50 LOC

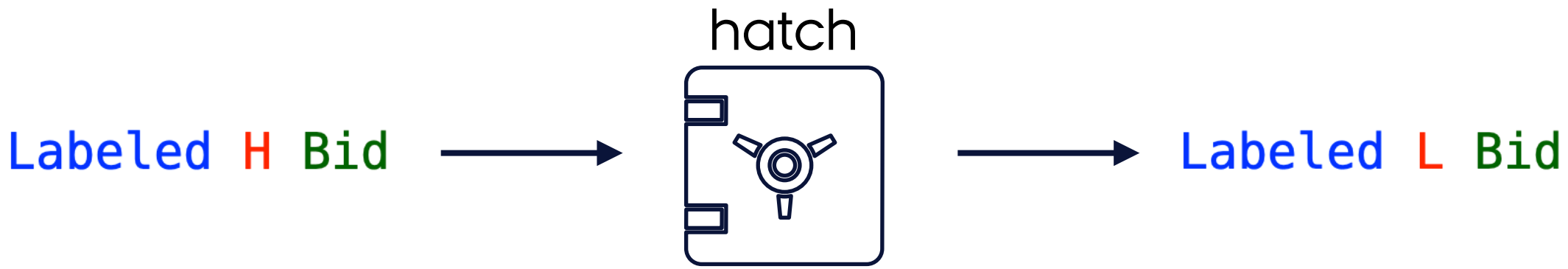
ESCAPE HATCHES (Sabelfeld and Myers, 2003)



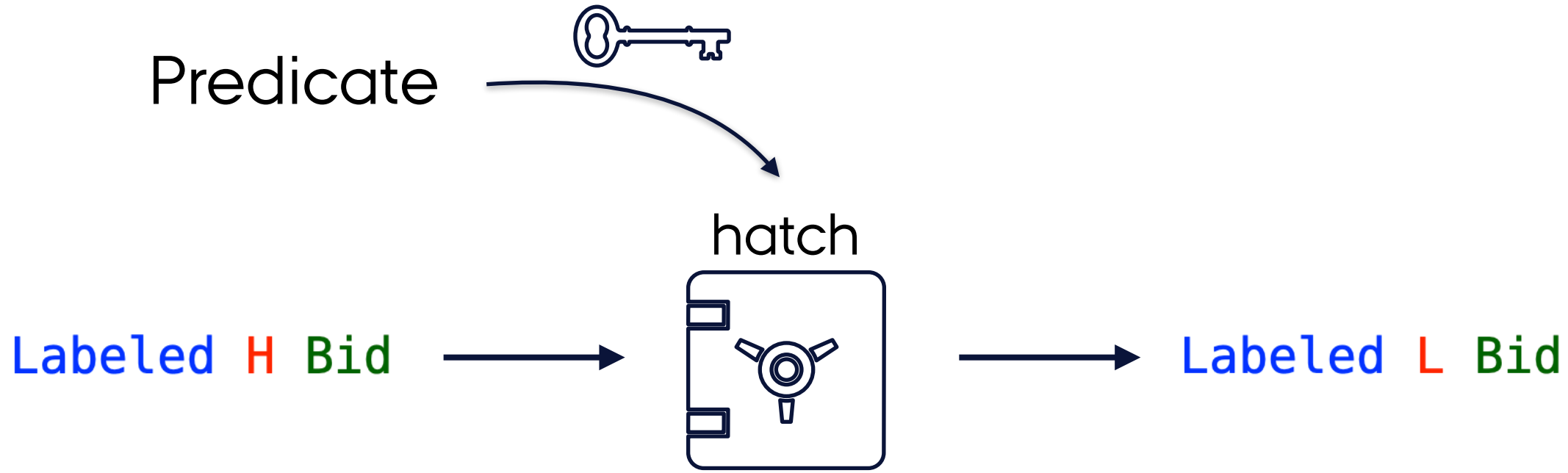
ESCAPE HATCHES (Sabelfeld and Myers, 2003)



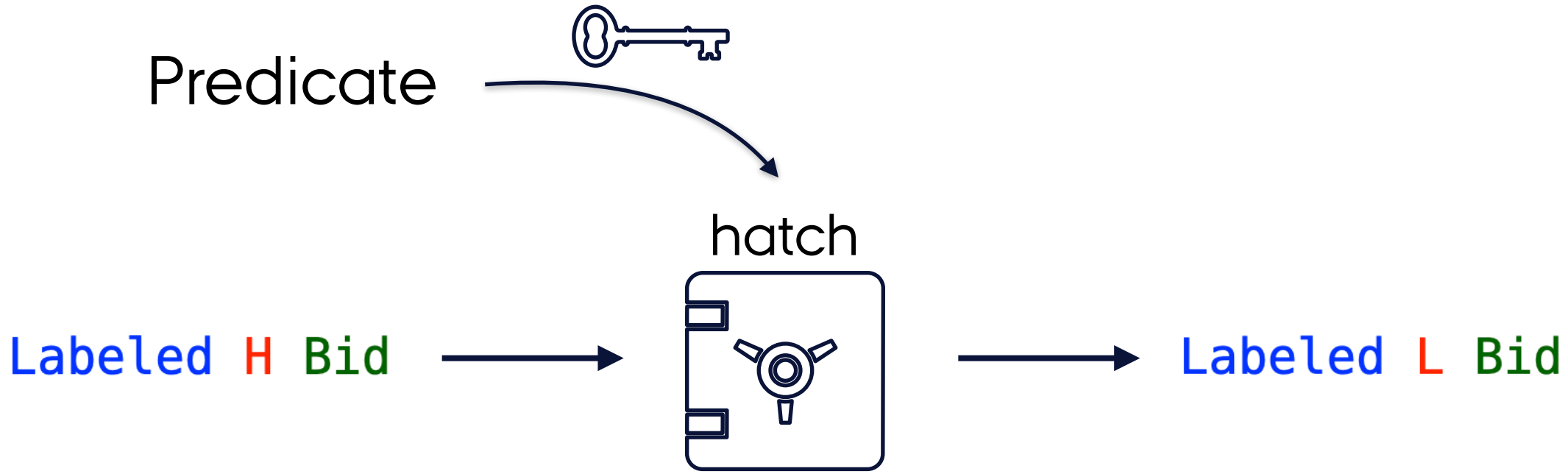
ESCAPE HATCHES (Sabelfeld and Myers, 2003)



ESCAPE HATCHES (Sabelfeld and Myers, 2003)



ESCAPE HATCHES (Sabelfeld and Myers, 2003)




```
Labeled H (b : Bid ** HighestBid rec b) -> Labeled L Bid
```

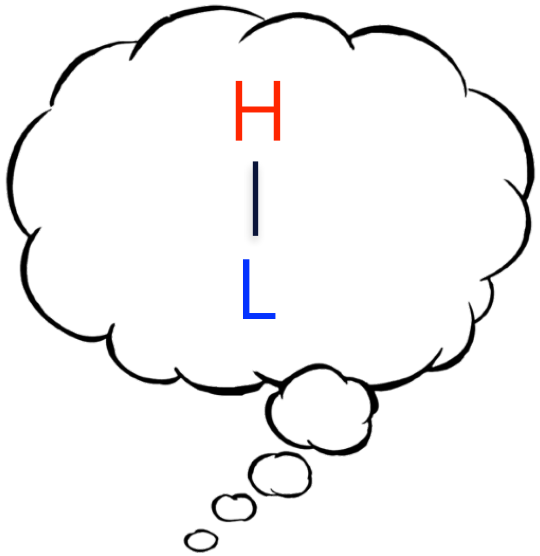
AUCTION III - SETUP

AUCTION III - SETUP

```
data Geq : Bid -> Labeled H Bid -> Type where
  -- ...

data MaxBid : Bid -> BidLog n -> Type where
  -- ...
```

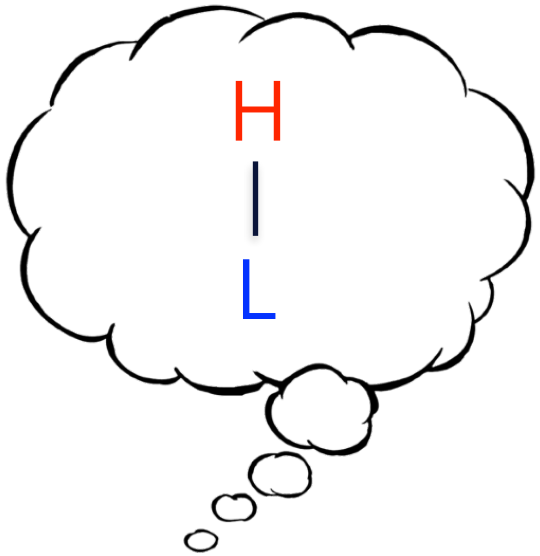
AUCTION III - SETUP

```
data Geq : Bid -> Labeled H Bid -> Type where
  -- ...

data MaxBid : Bid -> BidLog n -> Type where
  -- ...

HighestBid : BidLog _ -> Bid -> Type
HighestBid log b = (Elem (label b) log, MaxBid b log)
```



AUCTION III - SETUP

```
data Geq : Bid -> Labeled H Bid -> Type where
  -- ...

data MaxBid : Bid -> BidLog n -> Type where
  -- ...

HighestBid : BidLog _ -> Bid -> Type
HighestBid log b = (Elem (label b) log, MaxBid b log)

HighestBidHatch : BidLog _ -> Type
HighestBidHatch log =
  Labeled H (b : Bid ** HighestBid log b) -> Labeled L Bid
```

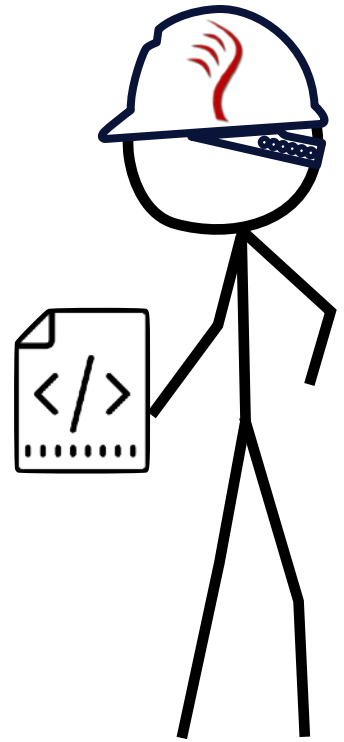


AUCTION III - SETUP

```
data Geq : Bid -> Labeled H Bid -> Type where
  -- ...

data MaxBid : Bid -> BidLog n -> Type where
  -- ...

HighestBid : BidLog _ -> Bid -> Type
HighestBid log b = (Elem (label b) log, MaxBid b log)

HighestBidHatch : BidLog _ -> Type
HighestBidHatch log =
  Labeled H (b : Bid ** HighestBid log b) -> Labeled L Bid

printBid : Labeled L Bid -> DIO L ()
```

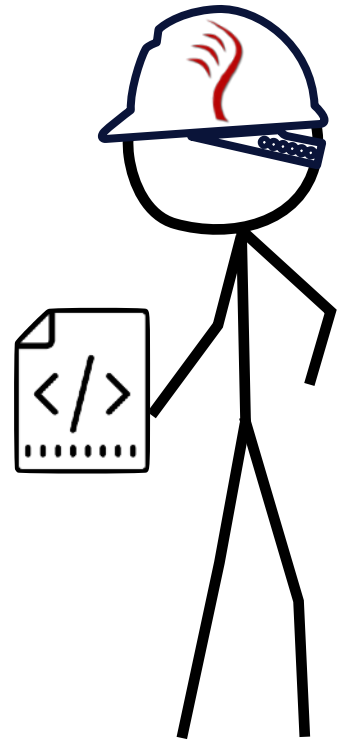
AUCTION III

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



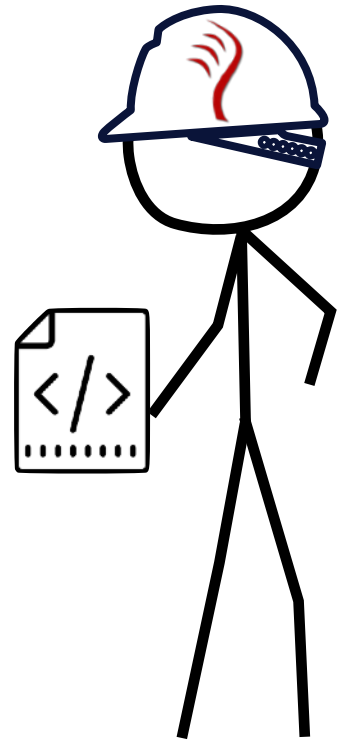
AUCTION III

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



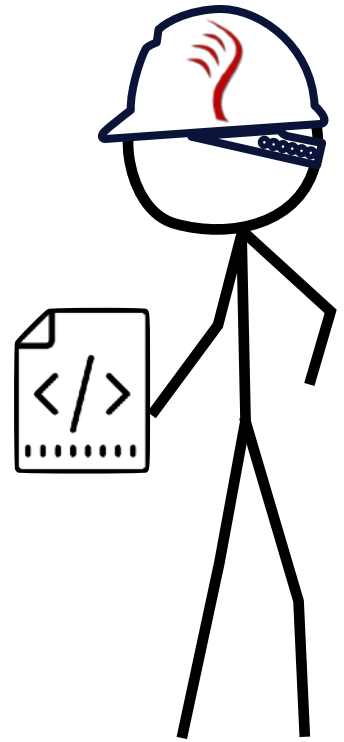
AUCTION III

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



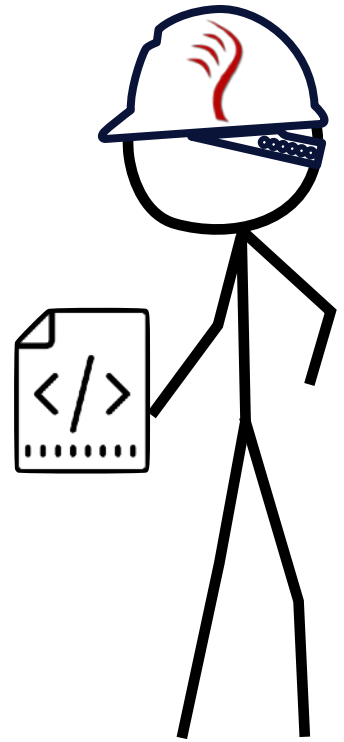
AUCTION III

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



AUCTION III

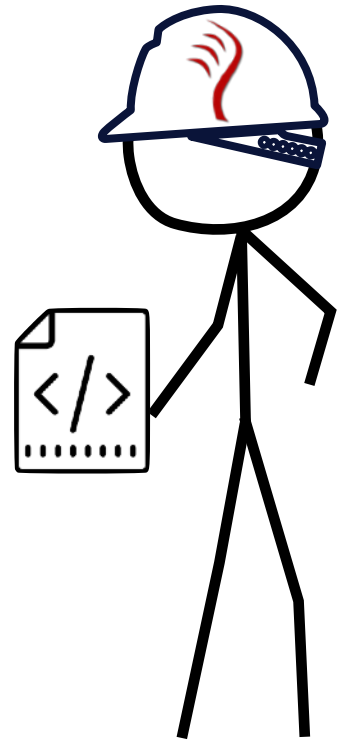
```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



AUCTION III

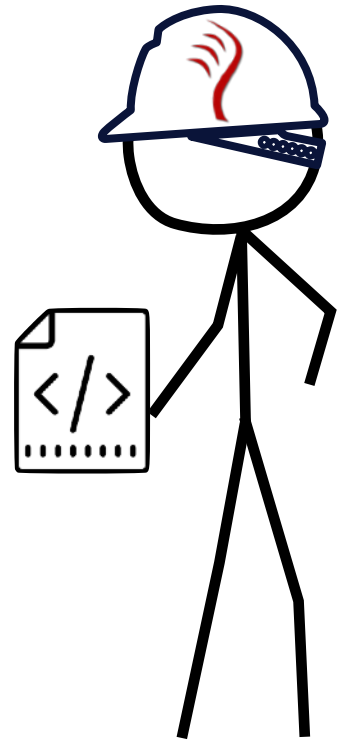
```
plug : Poset labelType
    => {l,l' : labelType}
    -> {auto flow : l `leq` l'}
    -> DIO l' a
    -> DIO l (Labeled l' a)
```

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



AUCTION III

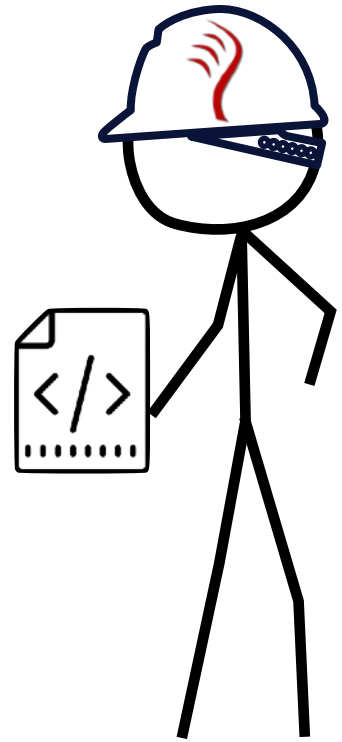
```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



AUCTION III

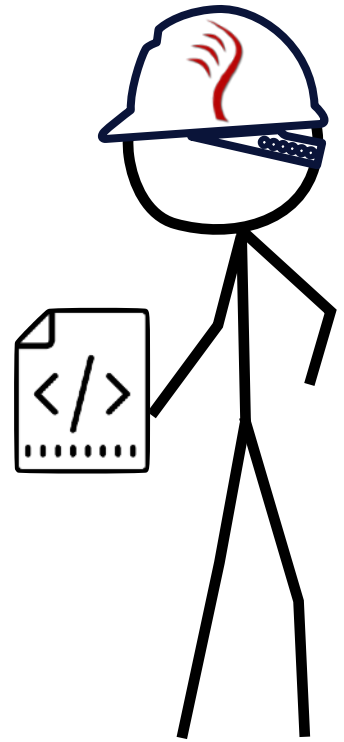
```
HighestBidHatch : BidLog _ -> Type
HighestBidHatch log =
  Labeled H (b : Bid ** HighestBid log b) -> Labeled L Bid
```

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



AUCTION III

```
auction : (log : BidLog (S n) ** HighestBidHatch log) -> DIO L ()
auction (bids ** hatch) =
     do max <- plug $ getMaxBid bids
        let max' : Labeled L Bid = hatch max
        printBid max'
  where
    getMaxBid : (r : BidLog (S n))
             -> DIO H (b : Bid ** HighestBid r b)
```



AUCTION III

```
main : IO ()
main =
  do putStrLn "##### Welcome to the auction! #####"
     putStrLn "Taking bids:"
     let log : BidLog _ = [!getBid, !getBid]
     run $ auction (predicateHatch log HighestBid)
     putStrLn "##### Bye bye! #####"
```

MORE EXAMPLES

- Time-based hatches
- Authority-based hatches
- Limiting hatch usage

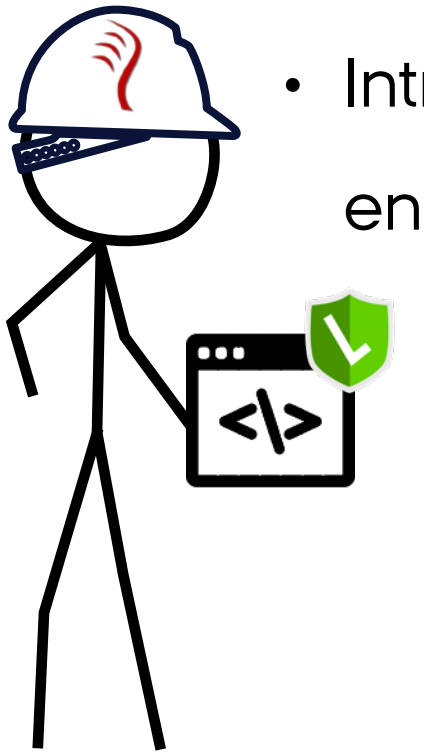
```
pwCheck : Labeled H Int
       -> DIO' L () [attempts ::: Attempts (3 + n) :-> Attempts n]
pwCheck secret =
  do x1 <- hatch secret 1
     x2 <- hatch secret 2
     x3 <- hatch secret 3
     x4 <- hatch secret 4 -- type error!
     pure ()
```

SECURITY GUARANTEES

Theorem 1 (PINI). *If* $c_1 \approx_{\ell_A} c_2$, $c_1 \Downarrow c_1'$, *and* $c_2 \Downarrow c_2'$ *then* $c_1' \approx_{\ell_A} c_2'$.

CONCLUSION

- **DEPSEC**: a static information-flow control library for Idris
- Showcased how dependent types increase the expressiveness of state-of-the-art information-flow control libraries
- Introduced predicate (and token) hatches for statically enforced declassification

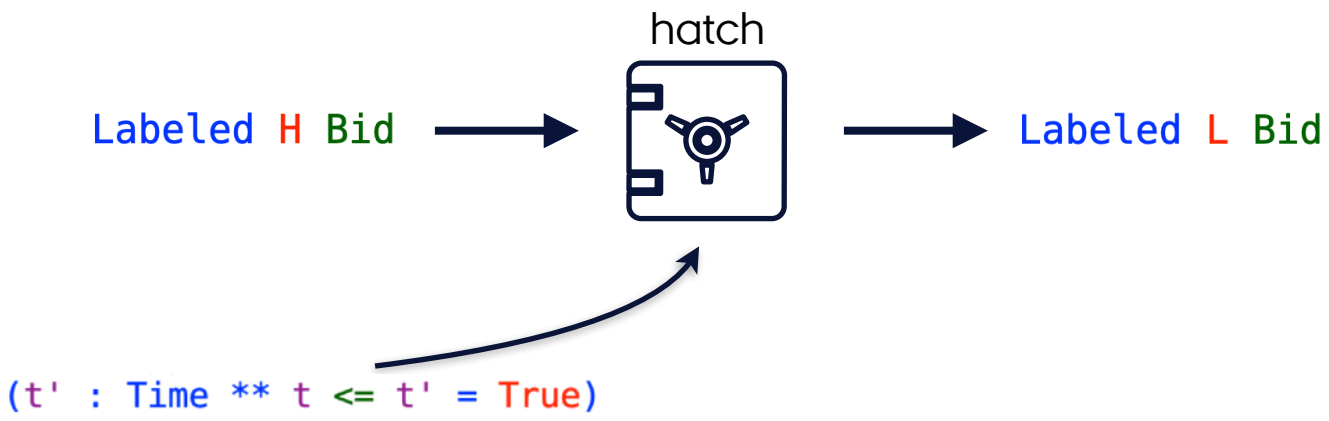


AUCTION IV

- Bids are secret when the auction is ongoing
- When the auction is done, all bids are public
- How to address **when** bids are declassified?

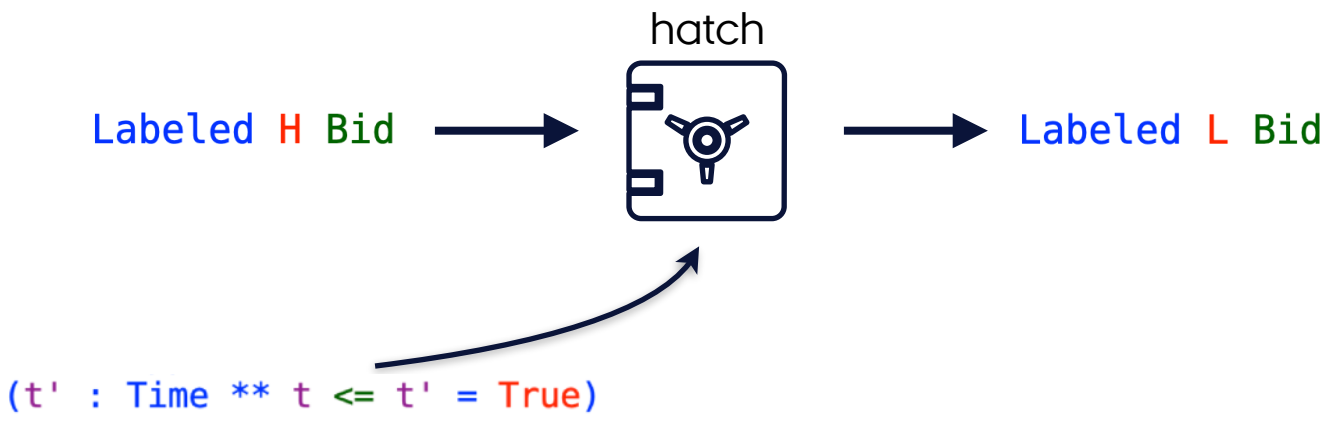
AUCTION IV - SETUP

AUCTION IV - SETUP



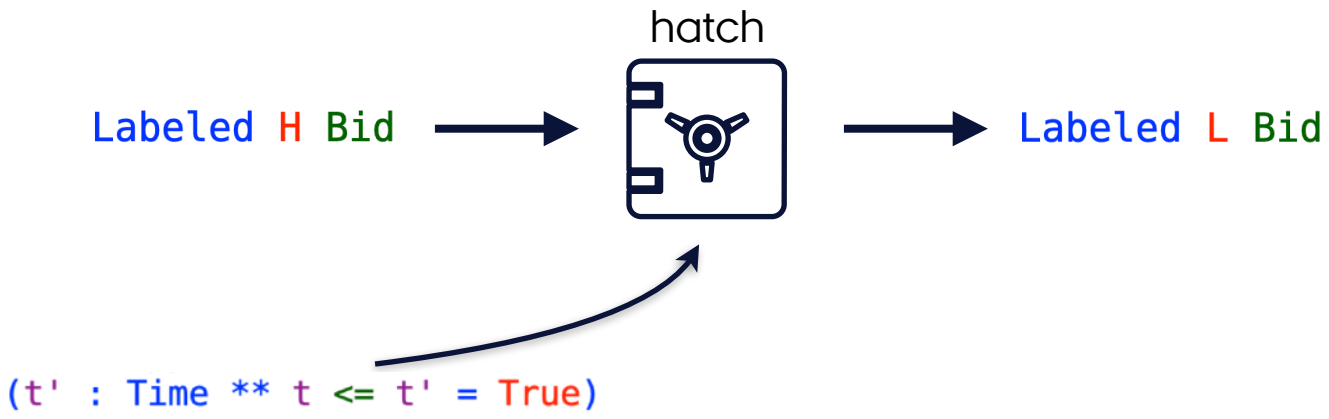
AUCTION IV - SETUP

```
data Time : Type where
  MkTime : (t : Integer) -> Time -- TCB
```



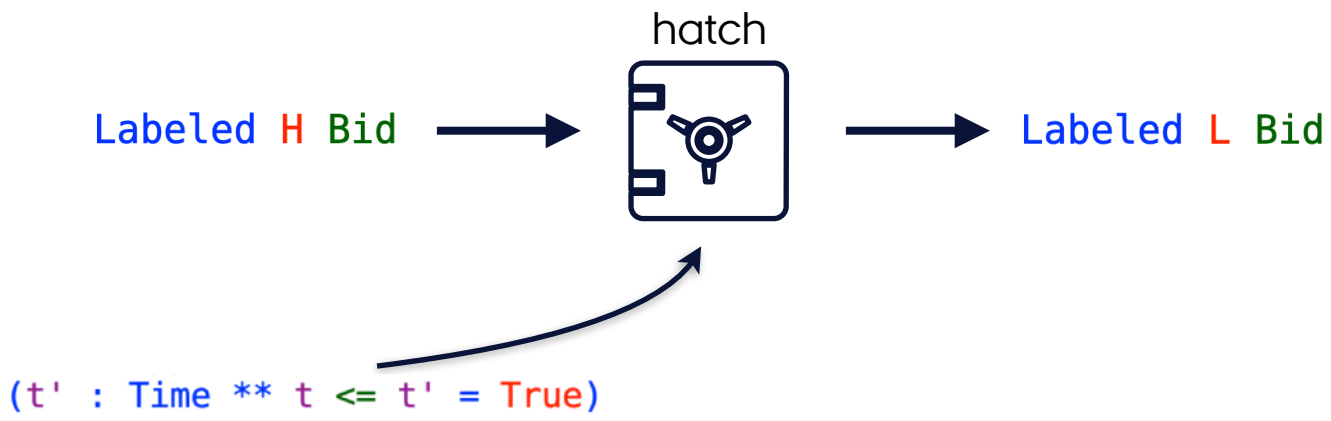
AUCTION IV - SETUP

```
data Time : Type where
  MkTime : (t : Integer) -> Time -- TCB

tick : DIO l Time
```



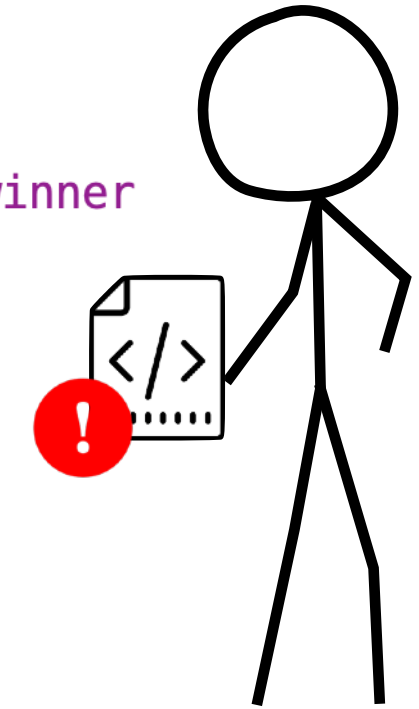
AUCTION IV - SETUP

```
data Time : Type where
  MkTime : (t : Integer) -> Time -- TCB

tick : DIO l Time

TimeHatch : Time -> Type
TimeHatch t = (t' : Time ** t <= t' = True) -> Labeled H Bid -> Labeled L Bid
```



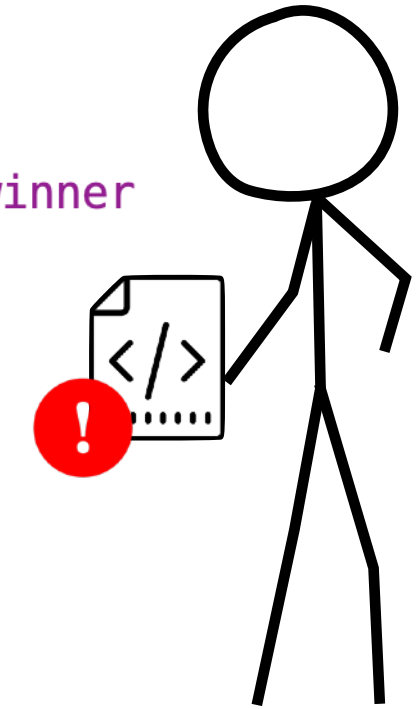
AUCTION IV

```
getWinner : BidLog (S n) -> TimeHatch t -> DIO L ()
getWinner log {t} hatch =
  do winner <- plug $ getMaxBid log
     time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Bid = hatch (time ** prf) winner
         in printBid declassified
       No _ => pure () -- auction is not over yet
```



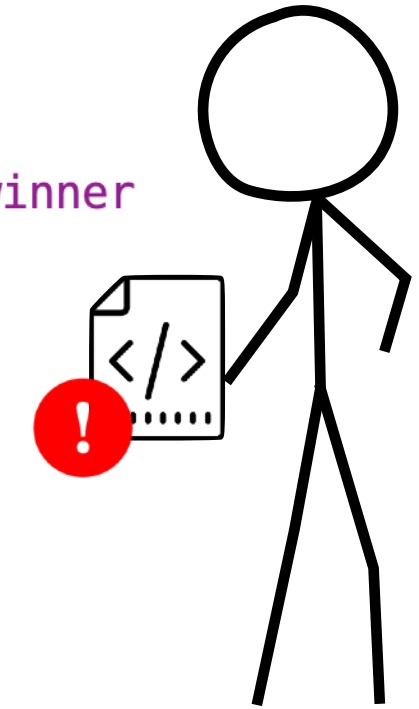
AUCTION IV

```
getWinner : BidLog (S n) -> TimeHatch t -> DIO L ()
getWinner log {t} hatch =
  do winner <- plug $ getMaxBid log
     time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Bid = hatch (time ** prf) winner
         in printBid declassified
       No _ => pure () -- auction is not over yet
```



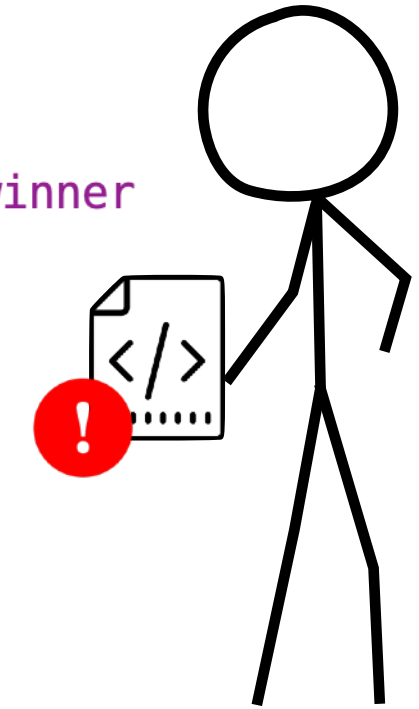
AUCTION IV

```
getWinner : BidLog (S n) -> TimeHatch t -> DIO L ()
getWinner log {t} hatch =
  do winner <- plug $ getMaxBid log
     time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Bid = hatch (time ** prf) winner
         in printBid declassified
       No _ => pure () -- auction is not over yet
```



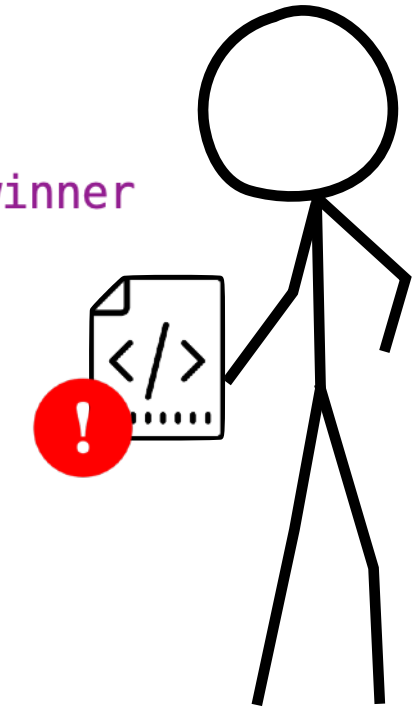
AUCTION IV

```
getWinner : BidLog (S n) -> TimeHatch t -> DIO L ()
getWinner log {t} hatch =
  do winner <- plug $ getMaxBid log
     time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Bid = hatch (time ** prf) winner
         in printBid declassified
       No _ => pure () -- auction is not over yet
```



AUCTION IV

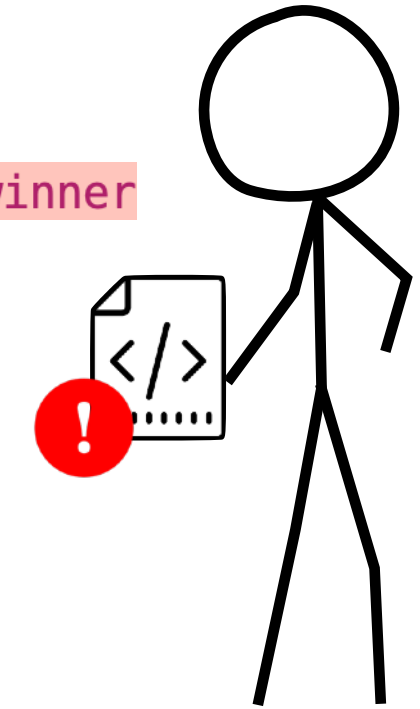
```
getWinner : BidLog (S n) -> TimeHatch t -> DIO L ()
getWinner log {t} hatch =
  do winner <- plug $ getMaxBid log
     time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Bid = hatch (time ** prf) winner
         in printBid declassified
       No _ => pure () -- auction is not over yet
```



AUCTION IV

```
TimeHatch : Time -> Type
TimeHatch t = (t' : Time ** t <= t' = True) -> Labeled H Bid -> Labeled L Bid
```

```
getWinner : BidLog (S n) -> TimeHatch t -> DIO L ()
getWinner log {t} hatch =
  do winner <- plug $ getMaxBid log
     time <- tick
     case decEq (t <= time) True of
       Yes prf =>
         let declassified : Labeled L Bid = hatch (time ** prf) winner
         in printBid declassified
       No _ => pure () -- auction is not over yet
```



HATCH BUILDERS

```
predicateHatch : Poset labelType
              => {l, l' : labelType}
              -> {D, E : Type}
              -> (d : D)
              -> (P : D -> E -> Type)
              -> (d : D ** Labeled l (e : E ** P d e) -> Labeled l' E)


tokenHatch : Poset labelType
          => {l, l' : labelType}
          -> {E, S : Type}
          -> (Q : S -> Type)
          -> (s : S ** Q s) -> Labeled l E -> Labeled l' E
```

LATTICE

```
interface JoinSemilattice a where
  join        : a -> a -> a
  commutative : (x, y : a) -> x `join` y = y `join` x
  associative : (x, y, z : a) -> x `join` (y `join` z) = (x `join` y) `join` z
  idempotent  : (x : a)       -> x `join` x = x

interface Poset a where
  leq           : a -> a -> Type
  reflexive     : (x : a) -> x `leq` x
  antisymmetric : (x, y : a) -> x `leq` y -> y `leq` x -> x = y
  transitive    : (x, y, z : a) -> x `leq` y -> y `leq` z -> x `leq` z
```