Kuifje

Quantitative Information Flow with Monads in Haskell

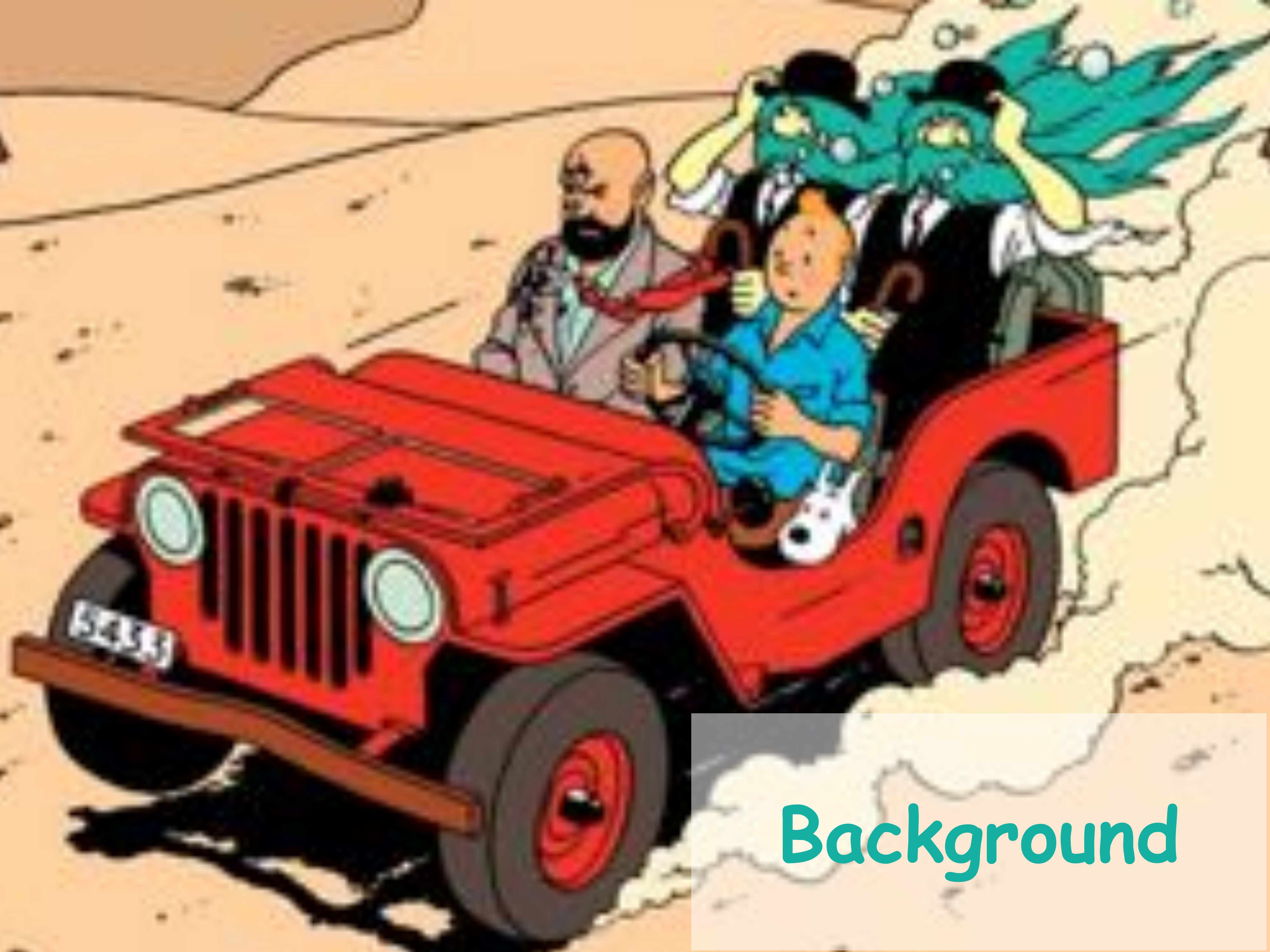
Joint work with

Carroll Morgan

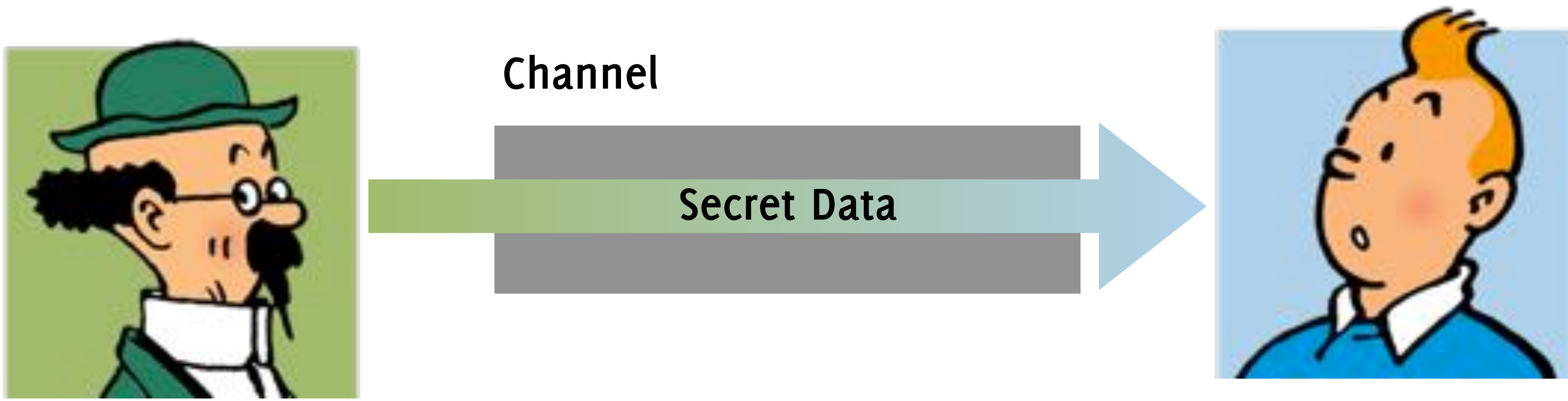
Annabelle McIver

Jeremy Gibbons

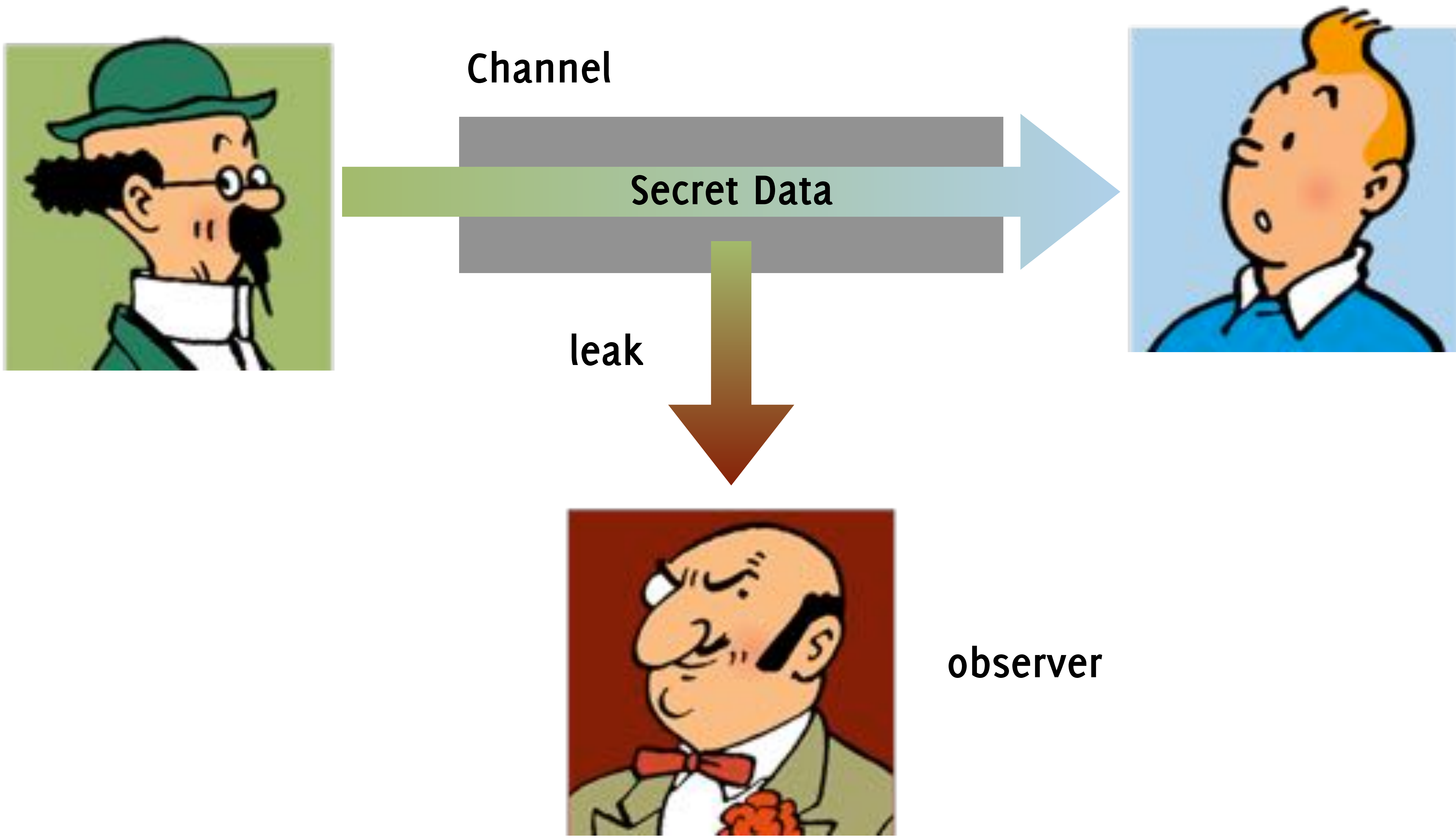
Background



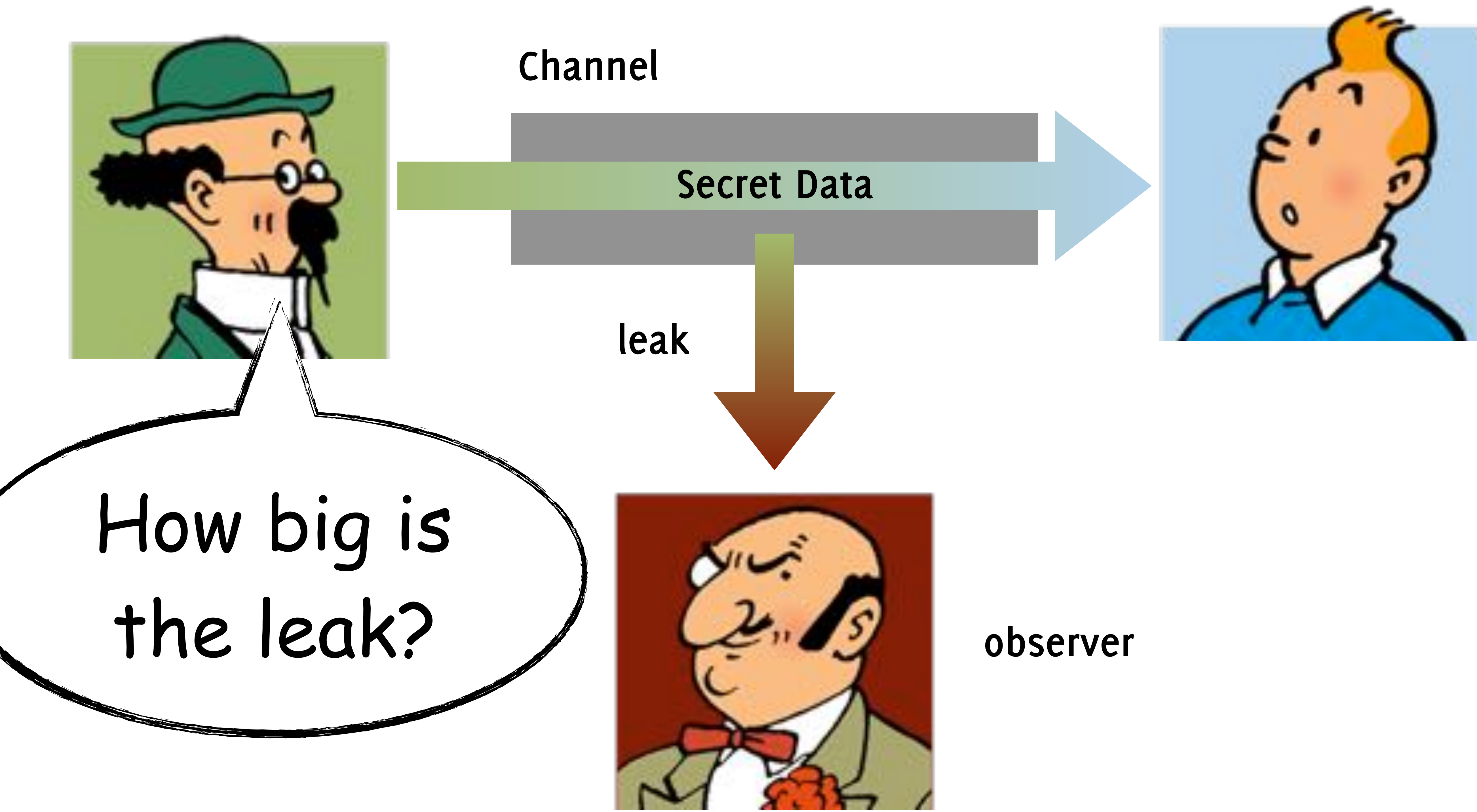
Quantified Information Flow



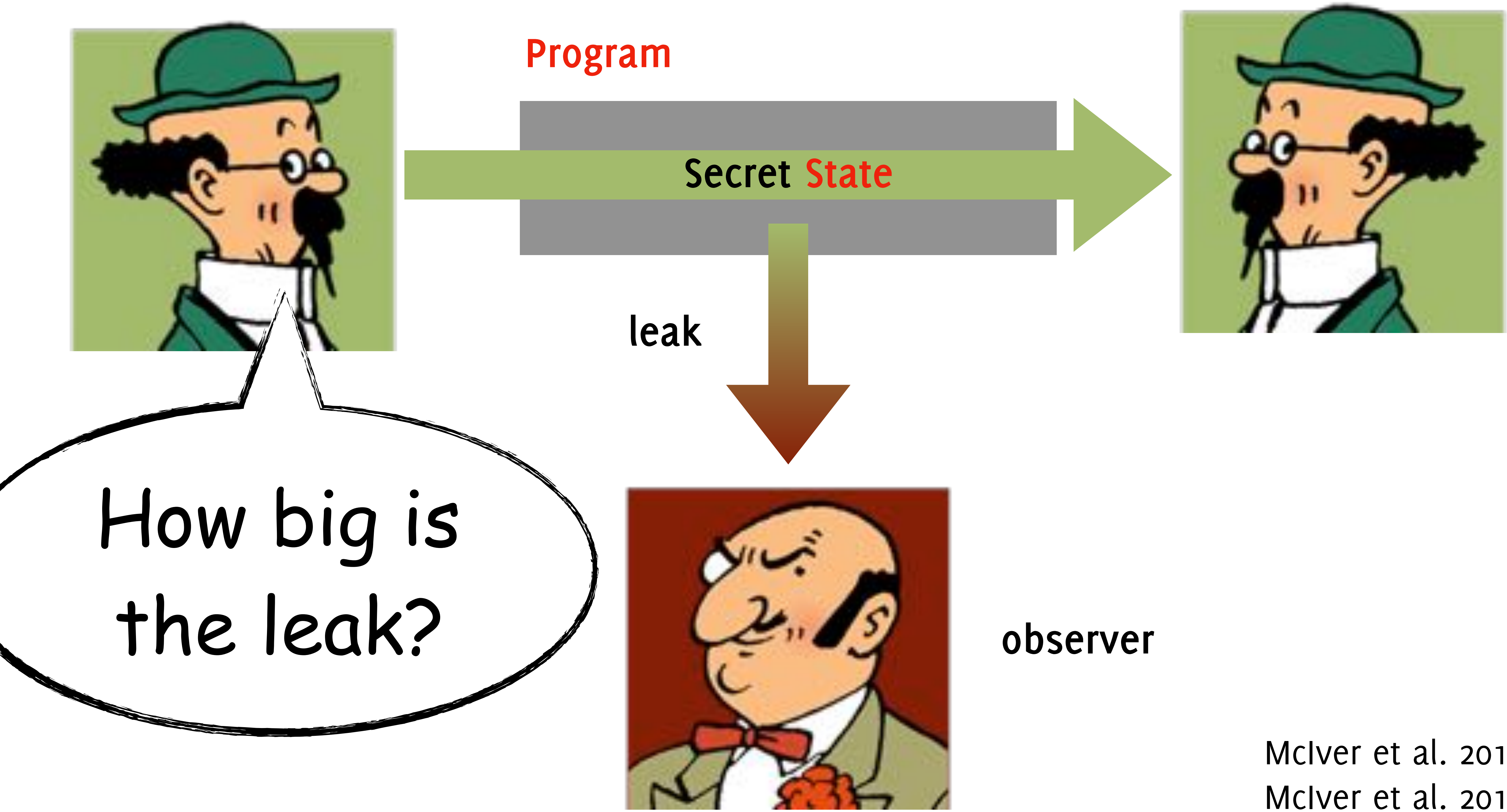
Quantified Information Flow



Quantified Information Flow

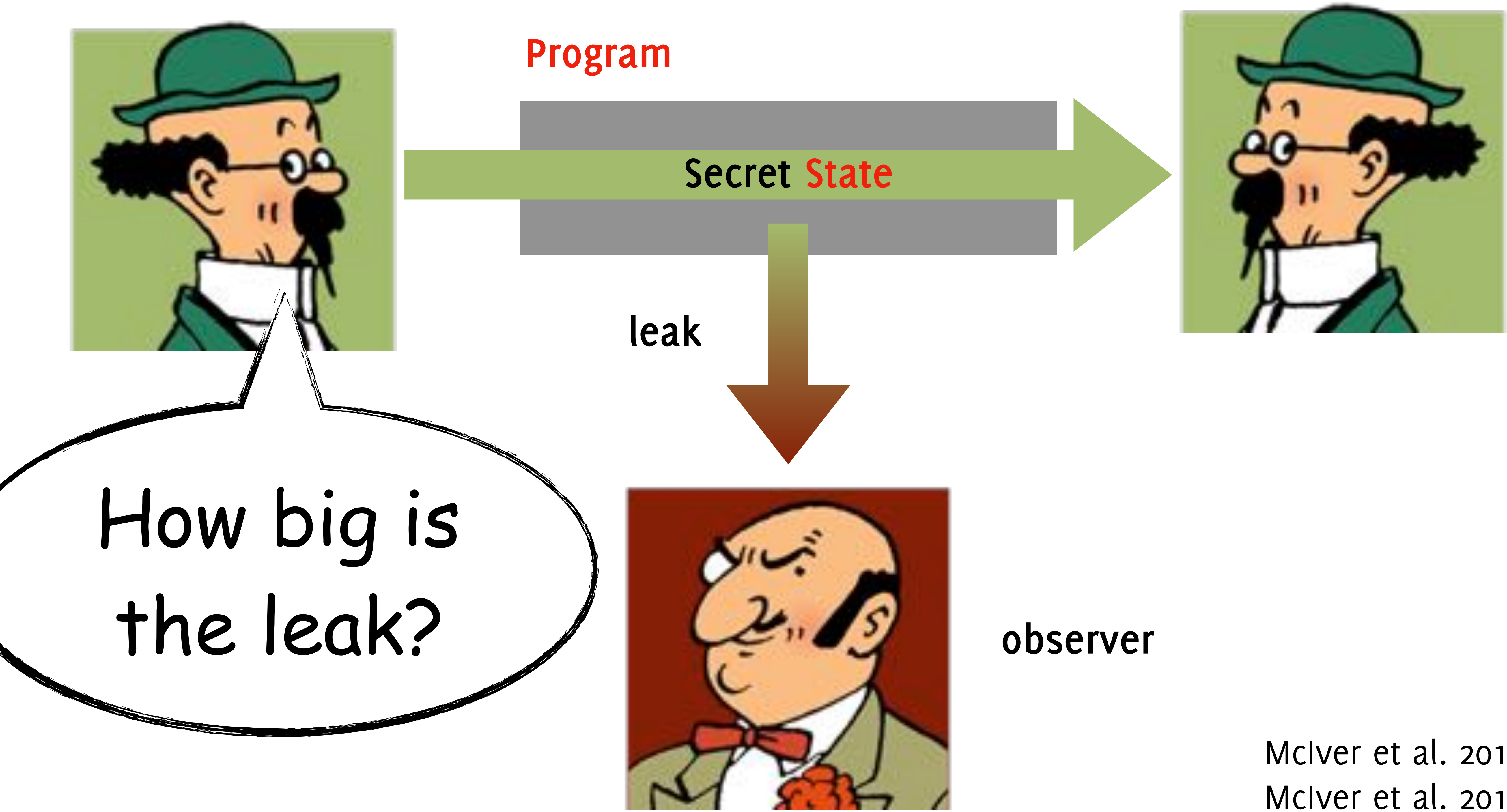


Quantified Information Flow



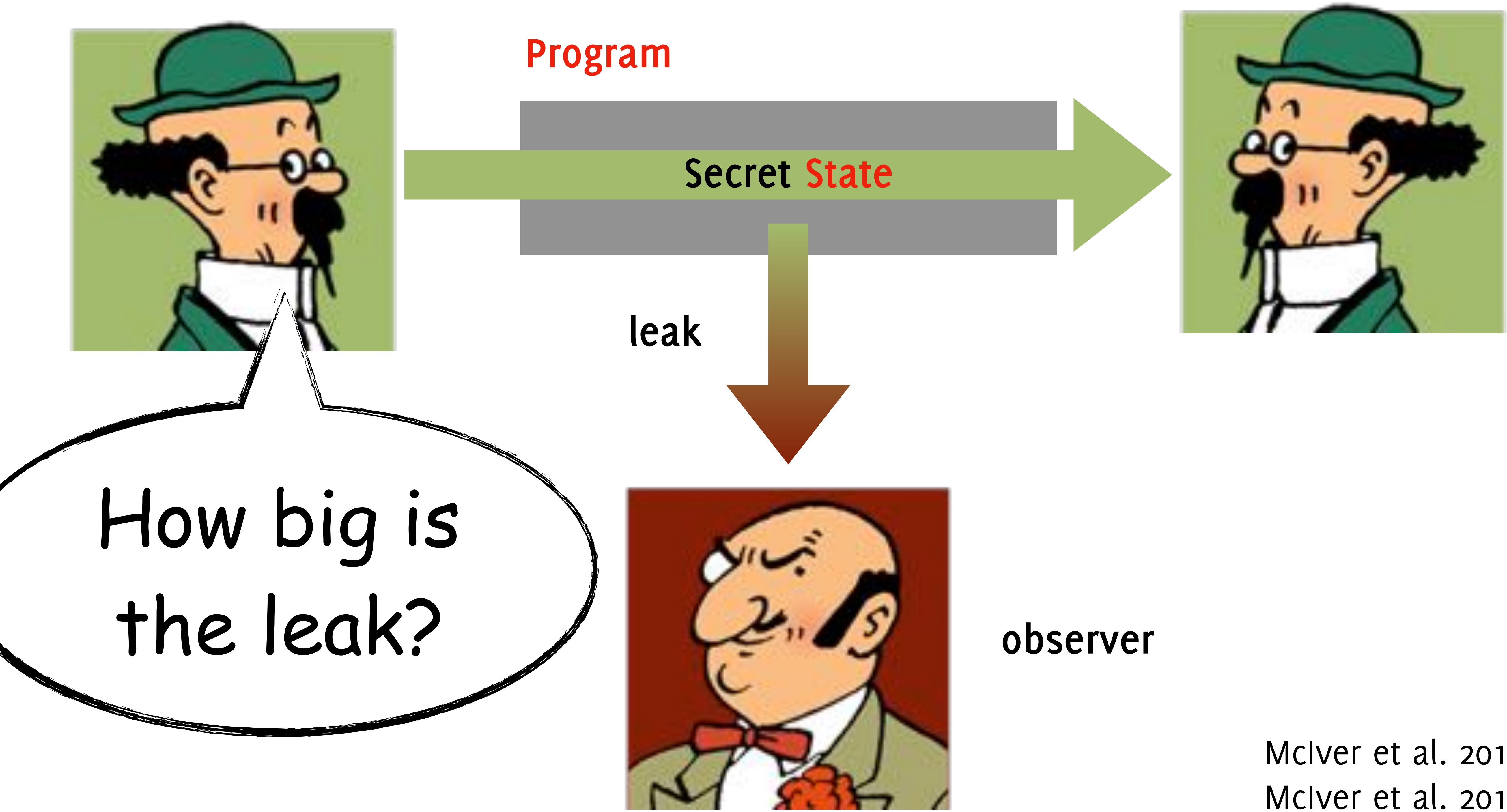
McIver et al. 2010
McIver et al. 2014

Quantified Information Flow



McIver et al. 2010
McIver et al. 2014

Quantified Information Flow



McIver et al. 2010
McIver et al. 2014

Kuifje

QIF-aware Haskell DSL

monad-based semantics

enables experiments

Outline

Syntax		**Semantics**

	Syntax		Semantics
1	CL s	$\xrightarrow{\text{sem}}$	s → s

Outline

	Syntax		Semantics
1	$\mathtt{CL}\ \mathtt{s}$	$\xrightarrow{\mathtt{sem}}$	$\mathtt{s} \to \mathtt{s}$
	$\subseteq$		
2	$\mathtt{PCL}\ \mathtt{s}$	$\xrightarrow{\mathtt{psem}}$	$\mathtt{s} \to \mathtt{D}\ \mathtt{s}$

Outline

	Syntax		Semantics
1	`CL s`	$\xrightarrow{\texttt{sem}}$	`s → s`
	$\cap$		
2	`PCL s`	$\xrightarrow{\texttt{psem}}$	`s → D s`
	$\cap$		
3	`Kuifje s`	$\xrightarrow{\texttt{posem}}$	`s → D (Bits,s)`

Outline

	Syntax		Semantics
1	`CL s`	$\xrightarrow{\texttt{sem}}$	`s` $\rightarrow$ `s`
	$\subseteq$		
2	`PCL s`	$\xrightarrow{\texttt{psem}}$	`s` $\rightarrow$ `D s`
	$\subseteq$		
3	`Kuifje s`	$\xrightarrow{\texttt{posem}}$	`s` $\rightarrow$ `D (Bits,s)`
4		$\xrightarrow{\texttt{hysem}}$	`D s` $\rightarrow$ `D (D s)`

Basic Command Language

Command Language

Version 101

```
type CL s = [Instruction s]

data Instruction s
  = Update (s → s)
  | If (s → Bool) (CL s) (CL s)
  | While (s → Bool) (CL s)
```

Command Language

without mutual recursion

```
data CL s
  -- []
  =  Skip
  -- Instruction s                   :  CL s
  |  Update (s → s)                  (CL s)
  |  If (s → Bool) (CL s)(CL s)      (CL s)
  |  While (s → Bool) (CL s)         (CL s)
```

Constructor Functions

```
skip :: CL s
skip = Skip

update :: (s → s) → CL s
update f = Update f skip

cond :: (s → Bool) → CL s → CL s → CL s
cond c p q = If c p q skip

while :: (s → Bool) → CL s → CL s
while c p = While c p skip
```

Sequential Composition

```
(⨾) :: CL s → CL s → CL s
         Skip ⨾ k = k
  Update f p ⨾ k = Update f (p ⨾ k)
  If c p q r ⨾ k = If c p q (r ⨾ k)
 While c p q ⨾ k = While c p (q ⨾ k)

instance Monoid (CL s) where
  mempty  = skip
  mappend = (⨾)
```

Example Program

```
data S = S { _x :: Int, _y :: Int}

example :: CL S
example =
  update (\s → s.^y $= 0) ;
  while (\s → s^.x > 0) (
    update (\s → s.^y $= (s^.y + s^.x)) ;
    update (\s → s.^x $= (s^.x - 1))
  )
```

Compositional Semantics

```
fold :: (CL_F s a → a) → (CL s → a)
```

where

```
data CL_F s r
  = Skip_F
  | Update_F (s → s) r
  | If_F (s → Bool) r r r
  | While_F (s → Bool) r r
```

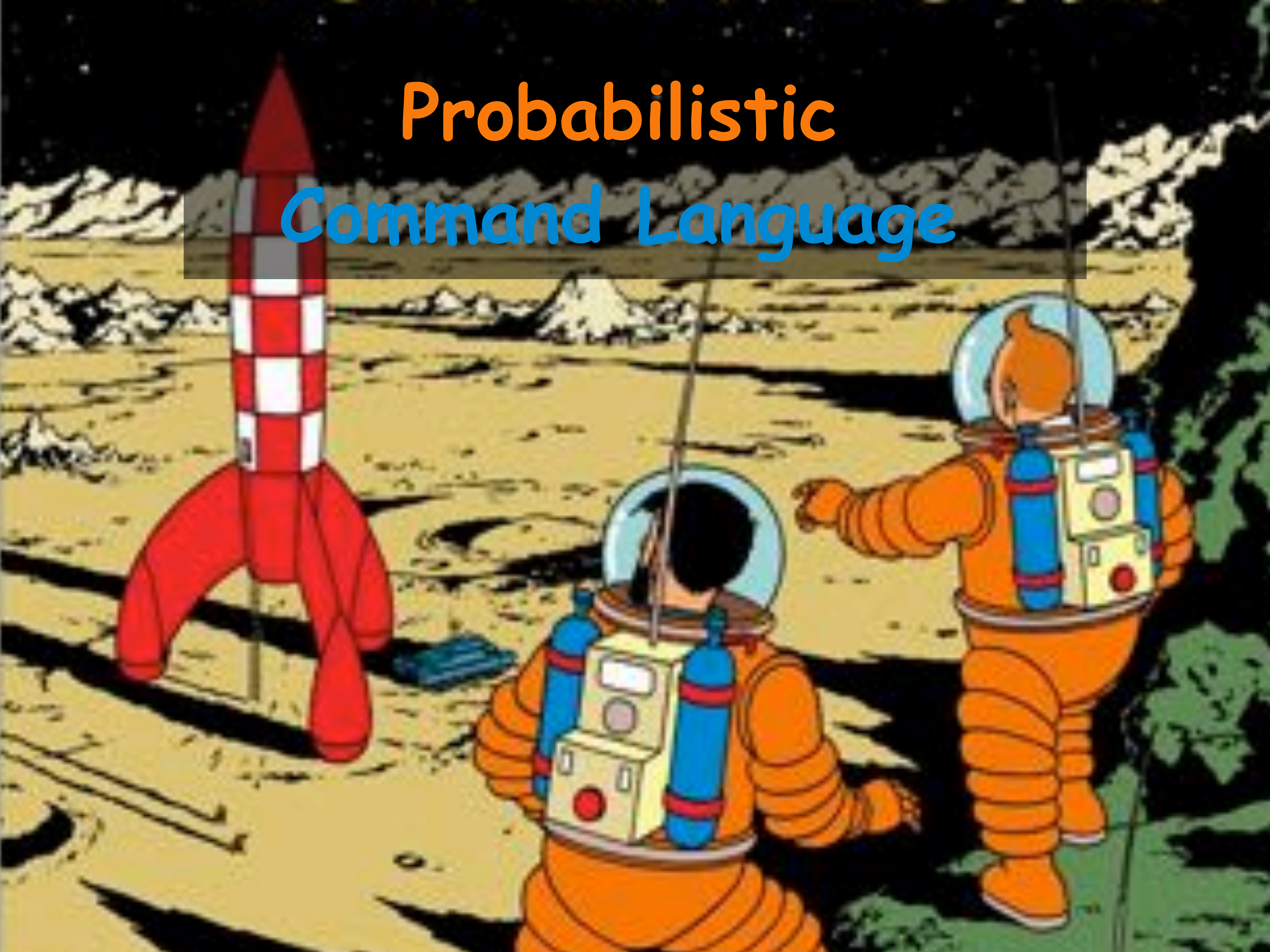
Semantics

```
sem :: CL s → (s → s)
sem = fold alg where
  alg :: CLF s (s → s) → (s → s)
  alg SkipF               = id
  alg (UpdateF f p)       = f >>> p
  alg (IfF c p q r)       = conditional c p q >>> r
  alg (WhileF c p q) =
    let while = conditional c (p >>> while) q
    in  while

  conditional :: (s → Bool) → (s → s)
              → (s → s) → (s → s)
  conditional c t e =
    (c &&& id) >>>
    (\(b,s) → if b then t s else e s)
```

Monoid Morphism

```
sem skip = id

sem (p ⨾ q) = sem p >>> sem q
```

Probabilistic Command Language



Discrete Distribution Monad

```
type Prob = Rational

newtype D a = D {runD :: [(a,Prob)]}

instance Monad D where
  return x = D [(x,1)]
  m >>= f  = D [(y, p * q)
               |(x, p) <- runD m
               ,(y, q) <- runD (f x)
               ]
```

Discrete Distribution Monad

```
type Prob = Rational

newtype D a = D {runD :: [(a,Prob)]}

uniform :: [a] → D a
uniform l = D [(x, 1 / length l)
              | x ← l
              ]

_ _⊕_ :: a → Prob → a → D a
x p⊕ y = D [(x, p), (y, 1 - p)]
```

PCL Syntax

```
type a ⇝ b = a → D b

data PCL s
  = Skip
  | Update (s ⇝ s)                     (PCL s)
  | If (s ⇝ Bool) (PCL s) (PCL s) (PCL s)
  | While (s ⇝ Bool) (PCL s)           (PCL s)
```

Example Program

```
data S = S { _x :: Int, _y :: Int}

example :: PCL S
example =
  update (\s → return (s.^y $= 0)) ⨟
  while (\s → return (s^.x > 0)) (
    update (\s → return (s.^y $= (s^.y + s^.x))) ⨟
    update (\s → (s.^x $= (s^.x - 1)) 2÷3⊕
                 (s.^x $= (s^.x - 2))
           )
  )
```

Semantics

```
psem :: PCL s → (s ⇝ s)
psem = fold alg where
 alg :: PCLF s (s ⇝ s) → (s ⇝ s)
 alg SkipF             = return
 alg (UpdateF f p)     = f ⟫ p
 alg (IfF c p q r)     = conditional c p q ⟫ r
 alg (WhileF c p q) =
   let while = conditional c (p ⟫ while) q
   in  while

conditional :: (s ⇝ Bool) → (s ⇝ s)
             → (s ⇝ s) → (s ⇝ s)
conditional c t e =
  (c &&& return) ⟫
  (\(b,s) → if b then t s else e s)
```

Monoid Morphism

```
psem skip = return

psem (p ⨟ q) = psem p ⟫= psem q
```

CL vs PCL

$$
\begin{tikzcd}
 & \textbf{Syntax} & \textbf{Semantics} \\
\textbf{Basic} & \mathtt{CL\ s} \arrow[r, "\mathtt{sem}"] \arrow[d, "\mathtt{embed}"'] & \mathtt{s} \to \mathtt{s} \arrow[d, "\mathtt{s} \to \mathtt{return}"] \\
\textbf{Probabilities} & \mathtt{PCL\ s} \arrow[r, "\mathtt{psem}"'] & \mathtt{s} \to \mathtt{D\ s}
\end{tikzcd}
$$

Leaking Probabilistic Command Language

Syntax

```
data Kuifje s
  = Skip
  | Update (s ⇝ s)                                (Kuifje s)
  | If (s ⇝ Bool)(Kuifje s)(Kuifje s) (Kuifje s)
  | While (s ⇝ Bool) (Kuifje s)                  (Kuifje s)
  | Observe (s ⇝ Bits)                            (Kuifje s)

type Bits = [Bool]
```

Constructor Function

Yoneda lemma in action

```
observe :: ToBits a ⇒ a → Kuifje s
observe x = Observe (toBits x) skip

class ToBits a where
  toBits :: a → Bits
```

Example

```
p :: Kuijfe (Bool,Bool)
p = observe (\(b1,b2) → b1 ½⊕ b2)
```

Semantics

```
posem :: Kuifje s → (s ⇝_B s)

type a ⇝_B b
  = a → D (Bits, b)
  = a → WriterT Bits D b
```

Example

```
p :: Kuifje (Bool,Bool)
p = observe (\(b1,b2) → b1 1/2⊕ b2)

boolPairs =
  uniform [(b1,b2) | b1 ← [True,False]
                   , b2 ← [True,False]]

> boolPairs ≫= posem p
```

```
1 % 4   ([False],(False,False))
1 % 8   ([False],(False,True))
1 % 8   ([False],(True,False))
1 % 8   ([True],(False,True))
1 % 8   ([True],(True,False))
1 % 4   ([True],(True,True))
```

Monoid Morphism

```
posem skip = return

posem (p ⨟ q) = posem p >>= posem q
```

PCL vs Kuifje

$$
\begin{tikzcd}
\texttt{PCL s} \arrow[r, "\texttt{psem}"] \arrow[d, swap, "\texttt{embed}"] & \texttt{s} \to \texttt{D s} \arrow[d, "\texttt{s} \to \texttt{lift}"] \\
\texttt{Kuifje s} \arrow[r, swap, "\texttt{posem}"] & \texttt{s} \to \texttt{D (Bits, s)}
\end{tikzcd}
$$

PCL vs Kuifje

$$
\begin{tikzcd}
\texttt{PCL s} \arrow[r, "\texttt{psem}"] \arrow[d, "\texttt{embed}"'] & \texttt{s} \to \texttt{D s} \arrow[d, "\texttt{s} \to \texttt{lift}"] \\
\texttt{Kuifje s} \arrow[r, "\texttt{posem}"'] & \texttt{s} \to \texttt{D (Bits, s)}
\end{tikzcd}
$$

Example

```
p1 :: PCL Bool          p2 :: PCL Bool
p1 = skip               p2 =  cond id skip skip
```

```
     uniform [True,False] ≫= psem p1
  ═  uniform [True,False] ≫= psem p2
  ═  1÷2 True
     1÷2 False
```

Example

```
p1 :: PCL Bool        p2 :: PCL Bool
p1 = skip             p2 =  cond id skip skip

    uniform [True,False] ⋙= posem p1
 ═  1÷2 ([],True)
    1÷2 ([],False)
 ≠  uniform [True,False] ⋙= posem p2
 ═  1÷2 ([True], True)
    1÷2 ([False],False)
```

Example

```
p1 :: PCL Bool        p2 :: PCL Bool
p1 = skip             p2 =  cond id skip skip

     uniform [True,False] >>= posem p1
  =  1÷2 ([],True)
     1÷2 ([],False)
  ≠  uniform [True,False] >>= posem p2
  =  1÷2 ([True], True)
     1÷2 ([False],False)
```

Conditionals leak their condition!

Semantics

```
type a ⇝B b = a → WriterT Bits D b

posem :: Kuifje s → (s ⇝B s)
posem = fold alg where
 alg :: KuifjeF s (s ⇝B s) → (s ⇝B s)
 alg SkipF               = return
 alg (UpdateF f p)       = (lift . f) ⟹ p
 alg (IfF c p q r)       = conditional c p q ⟹ r
 alg (WhileF c p q) =
   let while = conditional c (p ⟹ while) q
   in  while
 alg (ObserveF f q) = obsem f ⟹ p
```

Semantics

```
obsem :: (s ⇝ Bits) → (s ⇝B s)
obsem f = f &&& return

conditional :: (s ⇝ Bool) → (s ⇝B s)
             → (s ⇝B s) → (s ⇝B s)
conditional c t e =
  ((lift . c) &&& return) ⟹
  (obsem (\(b,s) → return b)) ⟹
  (\(b,s) → if b then t s else e s)
```

Hyper Kuifje

Hyper Semantics

Syntax | **Semantics**

$$\begin{tikzcd} \texttt{Kuifje s} \arrow[r, "\texttt{posem}"] & \texttt{s} \to \texttt{D (Bits, s)} \arrow[d, "\texttt{post}"] \\ & \texttt{D s} \to \texttt{D (D s)} \end{tikzcd}$$

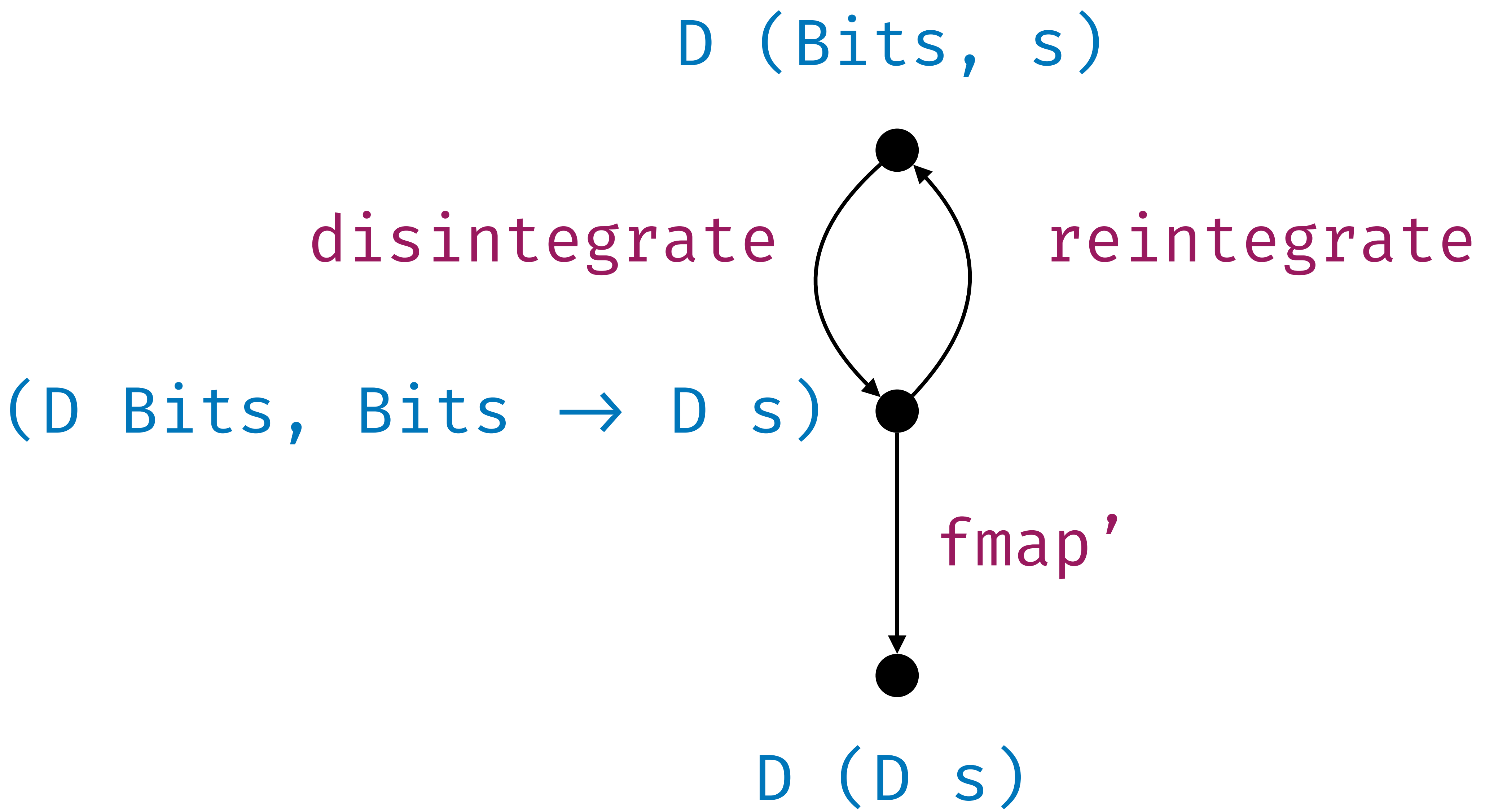
Leaked Values

Leaked Information

Hyper Semantics

```
hyper :: Ord s ⇒ Kuifje s → (D s → D (D s))
hyper = post . posem

post :: Ord s ⇒ (s → D (Bits, s))
             → (D s → D (D s))
post t =
 \d → fmap' (disintegrate (d ≫= t))
 where
  disintegrate
    :: D (a, b) → (D a, a → D b)

  fmap'
    :: Functor f ⇒ (f a, a → b) → f b
```

Lossy Post-Processing



Hyper Semantics

```
p :: Kuifje (Bool,Bool)
p = observe (\(b1,b2) → b1 1/2⊕ b2)

boolPairs =
  uniform [(b1,b2) | b1 ← [True,False]
                   , b2 ← [True,False]]
hyper p boolPairs
 :: D (D (Bool,Bool))
```

```
1÷2 1÷4 (False,True)
    1÷4 (True,False)
    1÷2 (True,True)
1÷2 1÷2 (False,False)
    1÷4 (False,True)
    1÷4 (True,False)
```

Fold Fusion

Syntax **Semantics**

$$\begin{tikzcd} \texttt{Kuifje s} \arrow[r, "\texttt{posem}"] & \texttt{s} \to \texttt{D (Bits, s)} \arrow[d, "\texttt{post}"] \\ & \texttt{D s} \to \texttt{D (D s)} \end{tikzcd}$$

Fold Fusion

Syntax **Semantics**

$$\begin{tikzcd} \mathtt{Kuifje\ s} \arrow[r, "\mathtt{fold\ alg}"] & \mathtt{s} \to \mathtt{D\ (Bits,\ s)} \arrow[d, "\mathtt{post}"] \\ & \mathtt{D\ s} \to \mathtt{D\ (D\ s)} \end{tikzcd}$$

Fold Fusion

Syntax **Semantics**

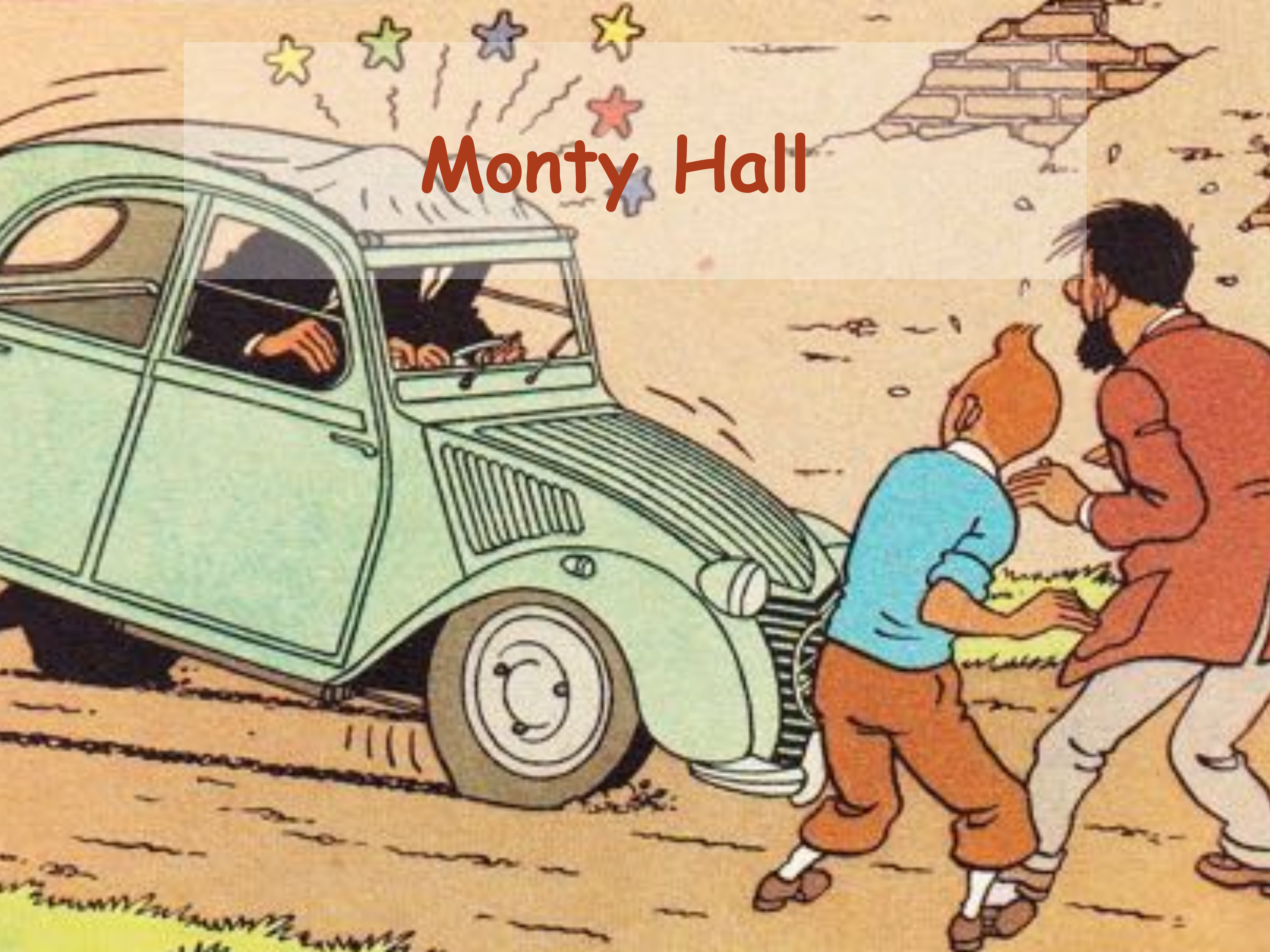
$$
\begin{tikzcd}
\texttt{Kuifje s} \arrow[r, "\texttt{fold alg}"] \arrow[dr, "\texttt{fold alg'}"'] & \texttt{s → D (Bits, s)} \arrow[d, "\texttt{post}"] \\
 & \texttt{D s → D (D s)}
\end{tikzcd}
$$

Fold Fusion

Syntax **Semantics**

$$
\begin{tikzcd}
\texttt{Kuifje s} \arrow[r, "\texttt{fold alg}"] \arrow[dr, "\texttt{hysem}"'] & \texttt{s} \to \texttt{D (Bits, s)} \arrow[d, "\texttt{post}"] \\
& \texttt{D s} \to \texttt{D (D s)}
\end{tikzcd}
$$

Monty Hall



Monty Hall

```
hall :: Door → Kuifje Door
hall chosenDoor =
  observe (\prizeDoor →
    uniform ([DoorA,DoorB,DoorC]
                  \\ [chosenDoor,prizeDoor]))
```

Monty Hall

```
hall :: Door → Kuifje Door
hall chosenDoor =
  observe (\prizeDoor →
    uniform ([DoorA,DoorB,DoorC]
               \\ [chosenDoor,prizeDoor]))

doors = uniform [DoorA,DoorB,DoorC]
monty = hysem (hall DoorA) doors
```

Monty Hall

```
hall :: Door → Kuifje Door
hall chosenDoor =
  observe (\prizeDoor →
    uniform ([DoorA,DoorB,DoorC]
                \\ [chosenDoor,prizeDoor]))

doors = uniform [DoorA,DoorB,DoorC]
monty = hysem (hall DoorA) doors
```

```
1÷2 1÷3 DoorA
    2÷3 DoorB
1÷2 1÷3 DoorA
    2÷3 DoorC
```

Bayes Vulnerability

Probability of a rational adversary guessing right when the distribution is known.

```
bv :: D a → Prob
bv d = maximum . fmap snd . runD
```

Conditional Entropy

```
condEntropy :: (D a → Prob)
            → (D (D a) → Prob)
condEntropy r h = weightedSum (fmap r h)
```

Conditional Entropy

```
condEntropy :: (D a → Prob)
            → (D (D a) → Prob)
condEntropy r h = weightedSum (fmap r h)
```

```
> condEntropy bv monty
2÷3
```

Fast Exponentiation

Fast Exponentiation

```
VAR B ← Base.                                     Global variables.
    E ← Exponent.
    p ← To be set to B^E.

BEGIN VAR b,e:= B,E                                Local variables.
  p:= 1
  WHILE e≠0 DO
    VAR r:= e MOD 2
    IF r≠0 THEN p:= p*b FI                         ← Side channel.
    b,e:= b^2,e÷2
  END
END
{ p = B^E }
```

Generalisation

Global variables.

```
VAR B ← Base.                                          Global variables.
    D ← Set of possible divisors.

    p ← To be set to B^E.
    E:= uniform(0..N-1)  Choose exponent uniformly at random

BEGIN VAR b,e:= B,E                                    Local variables.
  p:= 1
  WHILE e≠0 DO
    VAR d:= uniform(D) ← Choose divisor uniformly from set D.
    VAR r:= e MOD d
    IF r≠0 THEN p:= p*b^r FI                           ← Side channel.
    b,e:= b^d,e÷d
  END
END
{ p = B^E }   What does the adversary know about E at this point?
```

Evaluation

```
> condEntropy bv hyper2
```

```
> condEntropy bv hyper235
```

Evaluation

```
> condEntropy bv hyper2
1÷1
```

```
> condEntropy bv hyper235
```

Evaluation

```
> condEntropy bv hyper2
1÷1

> condEntropy bv hyper235
161÷1296
```

Conclusion

Kuifje

- QIF-aware Haskell DSL with
- hyper-distribution semantics
- featuring lots of FP patterns

Einde