This ain't your Daddy's Probability Monad

Haskell eXchange London

Lars Brünjes

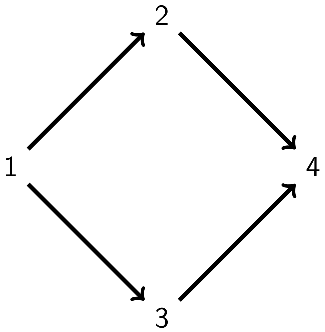
October 10, 2019

Motivation

- Traditional probability monads are great at modelling uncertainty.
- Things become more complicated when time enters the picture, especially in the presence of concurrency.
- Uncertainty, time and concurrency all need to be considered when trying to model the behavior of distributed systems.

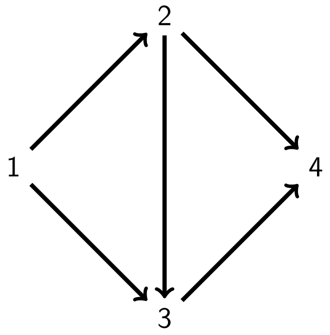
Example: The “Diamond”-Network

- We consider a network of four nodes, connected as indicated in the diagram.
- Each individual connection takes a time uniformly distributed between one second and two seconds and fails with a probability of 10%.
- What is the probability for a signal originating in node 1 to reach node 4? How long will it take?



Example: The "Diamond"-Network

- We consider a network of four nodes, connected as indicated in the diagram.
- Each individual connection takes a time uniformly distributed between one second and two seconds and fails with a probability of 10%.
- What is the probability for a signal originating in node 1 to reach node 4? How long will it take?
- How do time and probability change when an extra edge is added?



Modelling Uncertainty — MonadProb

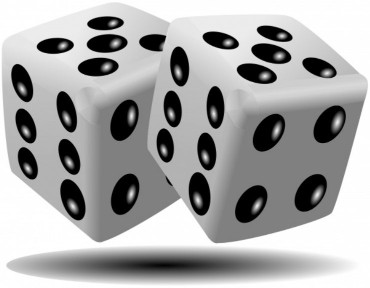
```haskell
type Prob p = (Ord p, Fractional p, Real p)

fromDouble :: (Fractional a, Real a) => Double -> a
fromDouble = fromRational . toRational
```

Modelling Uncertainty — MonadProb

```haskell
type Prob p = (Ord p, Fractional p, Real p)

fromDouble :: (Fractional a, Real a) => Double -> a
fromDouble = fromRational . toRational

class (Prob p, Monad m) => MonadProb p m | m -> p where

    coin :: p -> m Bool

    pick :: NonEmpty a -> m a
    pick (x :| [])       = return x
    pick (x :| y : ys) = do
        let len = length ys
            p   = recip $ fromIntegral $ len + 2
        b <- coin p
        if b then return x
             else pick $ y :| ys

unsafePick :: MonadProb p m => [a] -> m a
unsafePick = pick . fromList
```

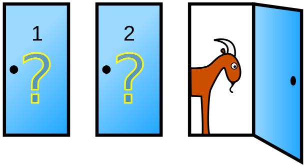
Example: Throwing Dice

```
die  ::  MonadProb p m => m Int
die  = pick $ 1 :| [2 .. 6]

dice  ::  MonadProb p m => Int -> m Int
dice  n
    | n <= 0     = return 0
    | otherwise  = sum <$> replicateM n die
```



Example: Monty Hall

```
data Prize = Goat | Car deriving (Show, Read, Eq, Ord)

data Strategy = Stay | Change deriving (Show, Read, Eq, Ord)

monty :: MonadProb p m => Strategy -> m Prize
monty s = do
    let doors    = 0 :| [1, 2]
    carDoor     <- pick doors
    let prizes   = (\i -> if i == carDoor then Car else Goat) <$> doors
    guess       <- pick doors
    openedDoor <- unsafePick $ filter (\i -> i /= guess && prizes !! i == Goat) doors
    let guess' = case s of
            Stay   -> guess
            Change -> head $ filter (\i -> i /= guess && i /= openedDoor) doors
    return $ prizes !! guess'
```



First Implementation: Sampling

```
newtype ProbS p m a = PS {runProbS :: m a }
    deriving (Functor, Applicative, Monad, MonadRandom)

instance (Prob p, MonadRandom m) => MonadProb p (ProbS p m) where
    coin p = do
        x <- fromDouble <$> getRandomR (0, 1)
        return $ x <= p
```

Trying Sampling

```haskell
diceS :: Int -> IO [Int]
diceS c = runProbS $ replicateM c (dice 2 :: ProbS Double IO Int)
```

```haskell
->>> diceS 40
      [6,6,8,9,6,7,8,5,7,5,9,5,5,6,11,8,3,7,9,8,10,9,6,9,10,8,9,4,3,10,11,2,7,11,6,6,4,6,7,7]
```

Trying Sampling

```haskell
diceS :: Int -> IO [Int]
diceS c = runProbS $ replicateM c (dice 2 :: ProbS Double IO Int)
```

```haskell
-- >>> diceS 40
--      [6,6,8,9,6,7,8,5,7,5,9,5,5,6,11,8,3,7,9,8,10,9,6,9,10,8,9,4,3,10,11,2,7,11,6,6,4,6,7,7]
```

```haskell
montyS :: Int -> Strategy -> IO [Prize]
montyS c s = runProbS $ replicateM c (monty s :: ProbS Double IO Prize)
```

```haskell
-- >>> montyS 15 Stay
-- [Goat,Goat,Goat,Car,Goat,Car,Goat,Goat,Goat,Car,Goat,Goat,Goat,Goat,Car]
-- >>> montyS 15 Change
-- [Goat,Car,Car,Car,Goat,Goat,Car,Car,Goat,Car,Goat,Goat,Car,Car,Car]
```

Second Implementation: Exact Distribution

```haskell
newtype ProbL p a = PL {runProbL :: [(a, p)]}
    deriving (Show, Read, Eq, Ord, Functor)
```

- A pair `(a, p)` means that the computation will have result `a` with probability `p`.
- The sum over all `p` is 1 (not expressed by the type).
- "Morally" we would like `Map a p`, but we do not have an `Ord`-instance for all `a`.

Second Implementation: Exact Distribution — Instances

```haskell
instance Prob p => Applicative (ProbL p) where
    pure = return
    (<*>) = ap

instance Prob p => Monad (ProbL p) where

    return a = PL [(a, 1)]

    PL aps >>= cont = PL $ do
        (a, p) <- aps
        (b, q) <- runProbL $ cont a
        return (b, p * q)
```

```haskell
instance Prob p => MonadProb p (ProbL p) where
    coin p
        | p <= 0    = return False
        | p >= 1    = return True
        | otherwise = PL [(True, p), (False, 1 - p)]

probLOrd :: (Prob p, Ord a) => ProbL p a -> Map a p
probLOrd = foldl' f Map.empty . runProbL
  where
    f m (a, p) = Map.insertWith (+) a p m
```

Trying Exact Distribution

```
diceL :: [(Int, Rational)]
diceL = Map.toList $ probLOrd $ dice 2
```

```
-->>> diceL
[(2,1 % 36),(3,1 % 18),(4,1 % 12),(5,1 % 9),(6,5 % 36),
    (7,1 % 6),(8,5 % 36),(9,1 % 9),(10,1 % 12),(11,1 % 18),(12,1 % 36)]
```

Trying Exact Distribution

```
diceL :: [(Int, Rational)]
diceL = Map.toList $ probLOrd $ dice 2
```

```
-->>> diceL
[(2,1 % 36),(3,1 % 18),(4,1 % 12),(5,1 % 9),(6,5 % 36),
    (7,1 % 6),(8,5 % 36),(9,1 % 9),(10,1 % 12),(11,1 % 18),(12,1 % 36)]
```

```
montyL :: Strategy -> [(Prize, Rational)]
montyL = Map.toList . probLOrd . monty
```

```
-->>> montyL Stay
[(Goat,2 % 3),(Car,1 % 3)]
>>> montyL Change
[(Goat,1 % 3),(Car,2 % 3)]
```

Adding Delays

```haskell
type Time t = (Fractional t, Ord t)

class ( MonadError () m
      , MonadProb p m
      , Time t
      ) => MonadDelay p t m | m -> p t where
    delay :: t -> m ()

absurd :: MonadDelay p t m => m a
absurd = throwError ()

catch :: MonadDelay p t m => m a -> m a -> m a
catch m h = catchError m $ const h
```

Approximating Uniform Delays

```haskell
uniform :: Ord t => MonadDelay p t m => t -> t -> Int -> m ()
uniform a b n
    | b < a      = absurd
    | b == a     = delay a
    | otherwise  = do
        let d = (b - a) / fromIntegral n
        t <- pick $ a :| [a + d * fromIntegral i | i <- [1 .. n]]
        delay t
```

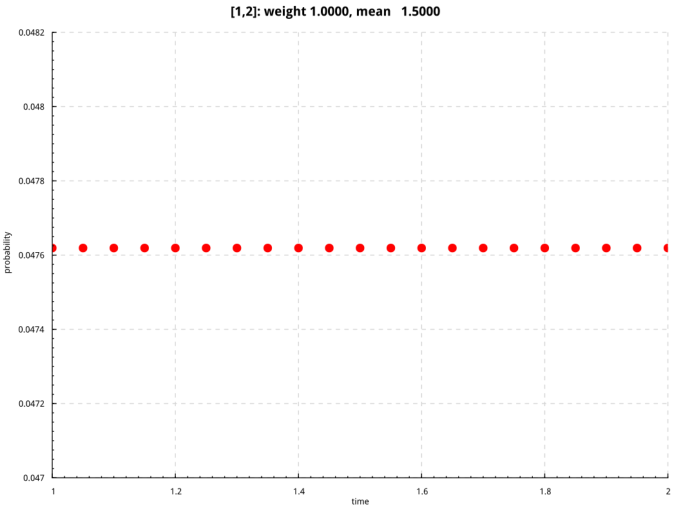
Approximating Uniform Delays

```
uniform :: Ord t => MonadDelay p t m => t -> t -> Int -> m ()
uniform a b n
    | b < a       = absurd
    | b == a      = delay a
    | otherwise   = do
        let d = (b - a) / fromIntegral n
        t <- pick $ a :| [a + d * fromIntegral i | i <- [1 .. n]]
        delay t
```

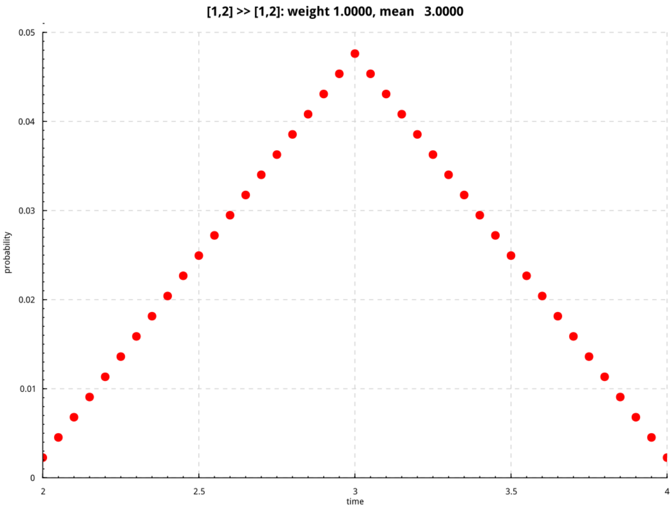
Remark

We could consider generalized delays instead to include "proper" uniform distributions, but they make implementing more difficult.

Example: `d = uniform 1 2 20`

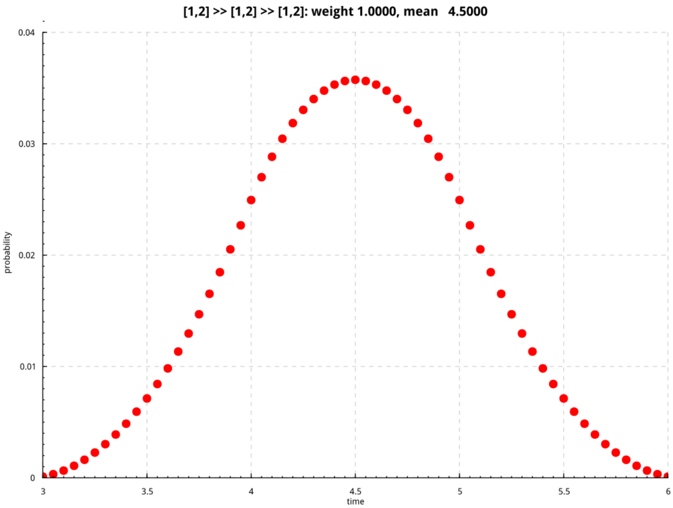


Example: `d >> d`



[1,2] >> [1,2]: weight 1.0000, mean 3.0000
0.05
0.04
0.03
0.02
0.01
0
probability
2
2.5
3
3.5
4
time

Example: `d >> d >> d`



[1,2] >> [1,2] >> [1,2]: weight 1.0000, mean 4.5000
0.04
0.03
0.02
0.01
0
probability
3
3.5
4
4.5
5
5.5
6
time

Just a Standard Transformer Stack

```
newtype DelayT p t m a = DT (WriterT (Sum t) (ExceptT () m) a)
    deriving (Functor, Applicative, Monad, MonadError (), MonadWriter (Sum t))

instance Time t => MonadTrans (DelayT p t) where
    lift  = DT . lift  .  lift

instance (Time t, MonadProb p m) => MonadProb p (DelayT p t m) where
    coin  =  lift  .  coin

instance (Time t, MonadProb p m) => MonadDelay p t (DelayT p t m) where
    delay  =  tell  .  Sum
```

Adding Concurrency

```haskell
class MonadDelay p t m => MonadRace p t m | m -> p t where
    race    ::  m a -> m b -> m (Either (a, m b) (m a, b))
```

Racing

We can "race" two computations and — after some time — get back the result of one and what remains of the other.

First-to-Finish Synchronization

```
ftf  :: MonadRace p t m => [m a] -> m a
ftf  []           = absurd
ftf  (ma : mas) = either  fst  snd <$> race ma (ftf mas)
```

Last-to-Finish Synchronization

```haskell
ltf  :: MonadRace p t m => [m a] -> m [a]
ltf  []           = return  []
ltf  (ma : mas) = do
    e <- race ma $ ltf mas
    case e of
        Left  (a,  m)   -> (a :) <$> m
        Right (m, xs)  -> m >>= \a -> return (a : xs)
```

Last-to-Finish Synchronization

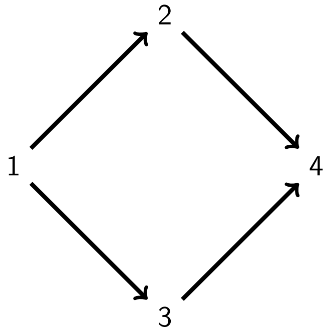
```haskell
ltf  :: MonadRace p t m => [m a] -> m [a]
ltf  []           = return  []
ltf  (ma : mas) = do
    e <- race ma $ ltf mas
    case e of
        Left  (a,  m)   -> (a :) <$> m
        Right (m, xs) -> m >>= \a -> return (a : xs)
```

Remark

We could use `ftf` and `ltf` as primitives instead of `race`. This would be strictly less powerfull, but on the other hand would allow for more complicated delays (not just discrete ones).

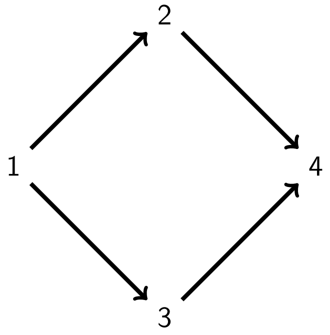
Implementing the “Diamond”-Network

```haskell
-- one edge
d :: (Ord t, Num t, MonadRace p t m) => m ()
d = do
        b <- coin 0.9
        if b then uniform 1 2 20 else absurd
```



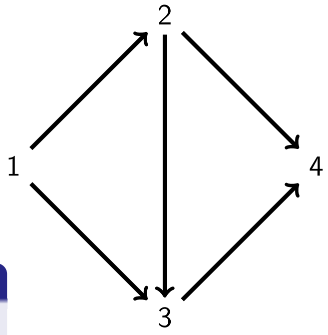
Implementing the "Diamond"-Network

```
diamond1 :: MonadRace p t m => m ()
diamond1 = do
    let d12 = d; d13 = d; d24 = d; d34 = d
    ftf [d12 >> d24, d13 >> d34]
```



Implementing the "Diamond"-Network

```
diamond2 :: MonadRace p t m => m ()
diamond2 = do
    let d12 = d; d13 = d; d23 = d; d24 = d; d34 = d
    e <- race d12 d13
    case e of
        Left  ((), r13)  -> ftf [d24, ftf [d23, r13] >> d34]
        Right (r12, ())  -> ftf [d34, r12 >> d24]
```



Remark

The full power of `race` is needed — `ftf` alone won't do!

First Implemention of `MonadRace`: Sampling

```haskell
newtype RaceS p t m a = RS {runRS :: m (Maybe (a, t))}
    deriving Functor
```

- A `Nothing` return-value corresponds to failure.
- A `Just (a, t)` return-value indicates result `a` after time `t`.

First Implemention of MonadRace: Sampling — Monad

```
instance (Time t, MonadRandom m) => Applicative (RaceS p t m) where
    pure = return
    (<*>) = ap

instance (Time t, MonadRandom m) => Monad (RaceS p t m) where
    RS x >>= cont = RS $ do
        mat <- x
        case mat of
            Nothing     -> return Nothing
            Just (a, t) -> do
                mbs <- runRS $ cont a
                case mbs of
                    Nothing     -> return Nothing
                    Just (b, s) -> return $ Just (b, t + s)
```

First Implemention of `MonadRace`: Sampling — `MonadError`

```haskell
instance (Time t, MonadRandom m) => MonadError () (RaceS p t m) where

    throwError ()  = RS $ return Nothing

    catchError (RS x) h = RS $ do
        mat <- x
        case mat of
            Nothing -> runRS $ h ()
            Just _  -> return mat
```

First Implemention of MonadRace: Sampling — MonadProb

```haskell
instance (Prob p, Time t, MonadRandom m) => MonadProb p (RaceS p t m) where
    coin p = RS $ do
        x <- fromDouble <$> getRandomR (0, 1)
        return $ Just (x <= p, 0)
```

First Implemention of `MonadRace`: Sampling — `MonadDelay`

```
instance (Prob p, Time t, MonadRandom m) => MonadDelay p t (RaceS p t m) where
    delay t = RS $ return $ Just ((), t)
```

First Implemention of MonadRace: Sampling — MonadRace

```haskell
instance (Prob p, Time t, MonadRandom m) => MonadRace p t (RaceS p t m) where
    race (RS x) (RS y) = RS $ do
        mat <- x
        mbs <- y
        case (mat, mbs) of
            (Nothing,    Nothing)    -> return Nothing
            (Just (a, t), Nothing)   -> return $ Just (Left (a, absurd), t)
            (Nothing,    Just (b, s)) -> return $ Just (Right (absurd, b), s)
            (Just (a, t), Just (b, s))
                | t <= s             -> return $ Just (Left (a, delay (s - t) >> return b), t)
                | otherwise          -> return $ Just (Right (delay (t - s) >> return a, b), s)
```

Pros and Cons of Sampling

- Sampling is straight forward and efficient.
- We could even support more sophisticated delays, like “proper” uniform delays easily.
- On the other hand, it would be nice to get exact results (at least for small examples).

Second Implemention of `MonadRace`: Exact Distribution

newtype **RaceL** p t a = **RL** {runRL :: [(t, p, a)]}
  deriving (**Show**, **Read**, **Eq**, **Ord**, **Functor**)

- A triple `(t, p, a)` corresponds to result `a` being obtained after time `t` with probability `p`.
- The sum of all `p` in the list, the weight, can be strictly smaller than 1, in which case there is a positive probability that the computation fails.
- "Morally" we would like `Map a (Map t p)`, but again we do not have an `Ord`-instance for all `a`.

Second Implemention of MonadRace: Exact Distribution — MonadError

```
instance (Prob p, Time t) => Monad (RaceL p t) where
    return a = RL [(0, 1, a)]
    RL xs >>= cont = RL $ do
        (ta, pa, a) <- xs
        (tb, pb, b) <- runRL $ cont a
        return (ta + tb, pa * pb, b)

instance (Prob p, Time t) => MonadError () (RaceL p t) where
    throwError () = RL []
    catchError (RL []) h = h ()
    catchError m       _ = m
```

```
instance (Prob p, Time t) => MonadProb p (RaceL p t) where
    coin p
        | p <= 0     = return False
        | p >= 1     = return True
        | otherwise  = RL [(0, p, True), (0, 1 - p, False)]
```

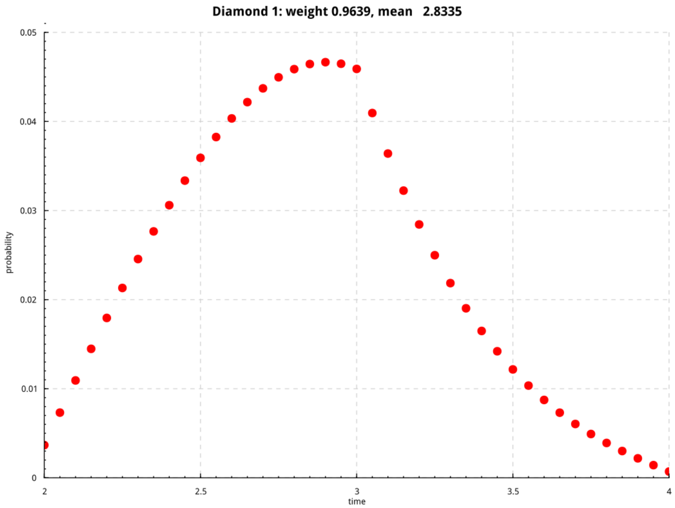
Second Implemention of MonadRace: Exact Distribution — MonadDelay

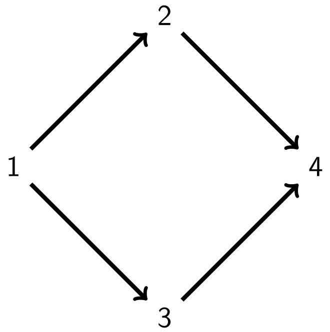
```
instance (Prob p, Time t) => MonadDelay p t (RaceL p t) where
    delay t = RL [(t, 1, ())]
```

Second Implemention of MonadRace: Exact Distribution — MonadRace

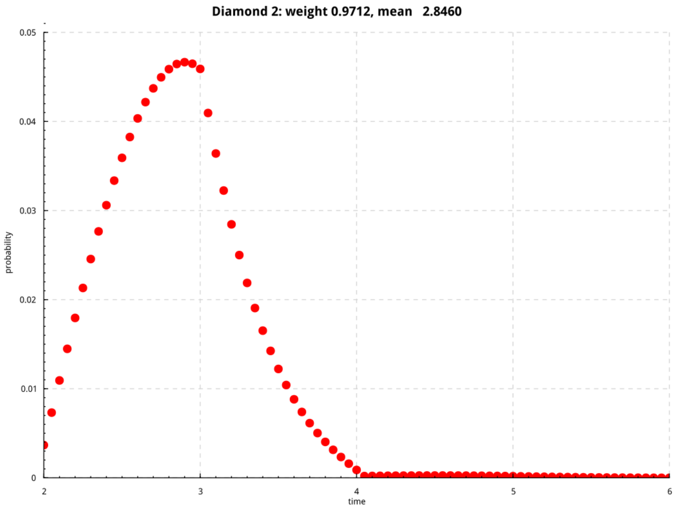
```
instance (Prob p, Time t) => MonadRace p t (RaceL p t) where
    race (RL xs) (RL ys) = RL $ do
        let fx  = max 0 $ min 1 $ 1 - weight xs
            fy  = max 0 $ min 1 $ 1 - weight ys
            za  = if fy > 0 then [(ta, pa * fy, Left (a, absurd)) | (ta, pa, a) <- xs] else []
            zb  = if fx > 0 then [(tb, pb * fx, Right (absurd, b)) | (tb, pb, b) <- ys] else []
            zab = do
                (ta, pa, a) <- xs
                (tb, pb, b) <- ys
                return $ if ta <= tb
                    then (ta, pa * pb, Left (a, RL [(tb - ta, 1, b)]))
                    else (tb, pa * pb, Right (RL [(ta - tb, 1, a)], b))
        za ++ zb ++ zab
      where
        weight :: [(t, p, c)] -> p
        weight ws = sum [p | (_, p, _) <- ws]
```

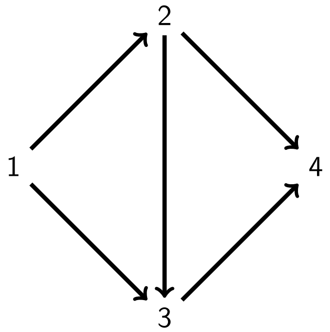
Example: The "Diamond"-Network





Example: The "Diamond"-Network





Thank you for your attention!

- EMail: `lars.bruenjes@iohk.io`
- Twitter: `@LarsBrunjes`
- GitHub: `https://github.com/brunjlar/`