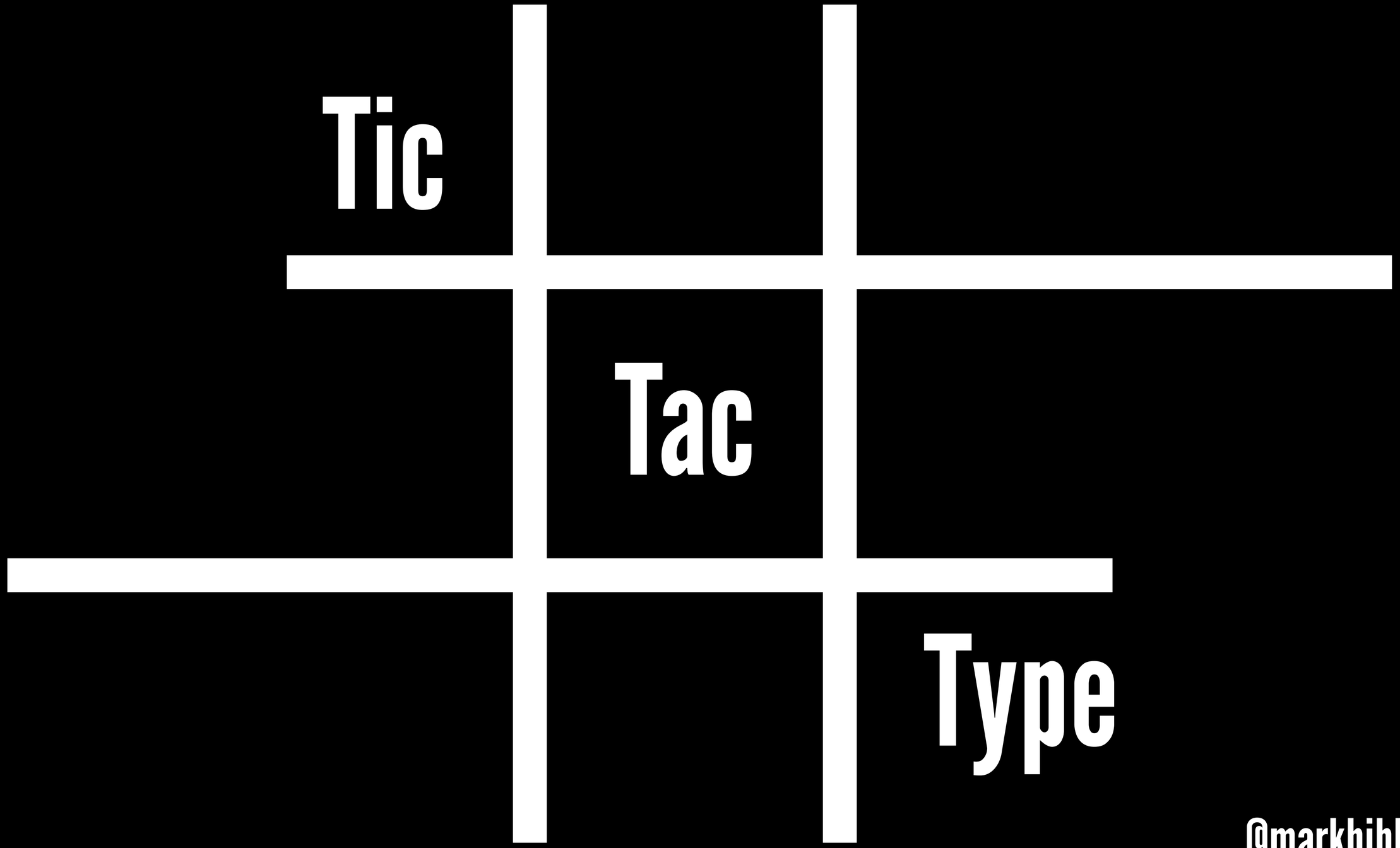
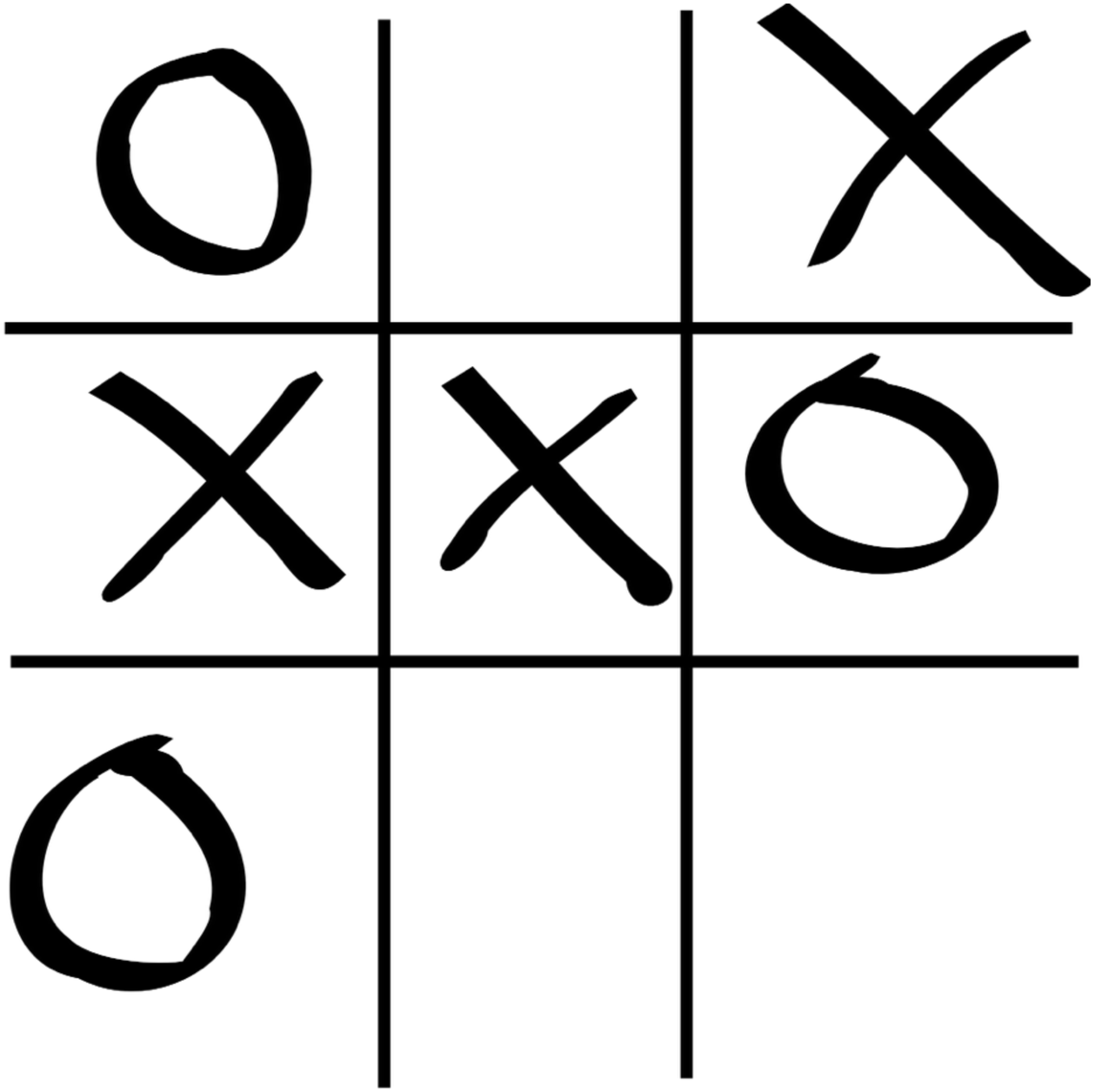


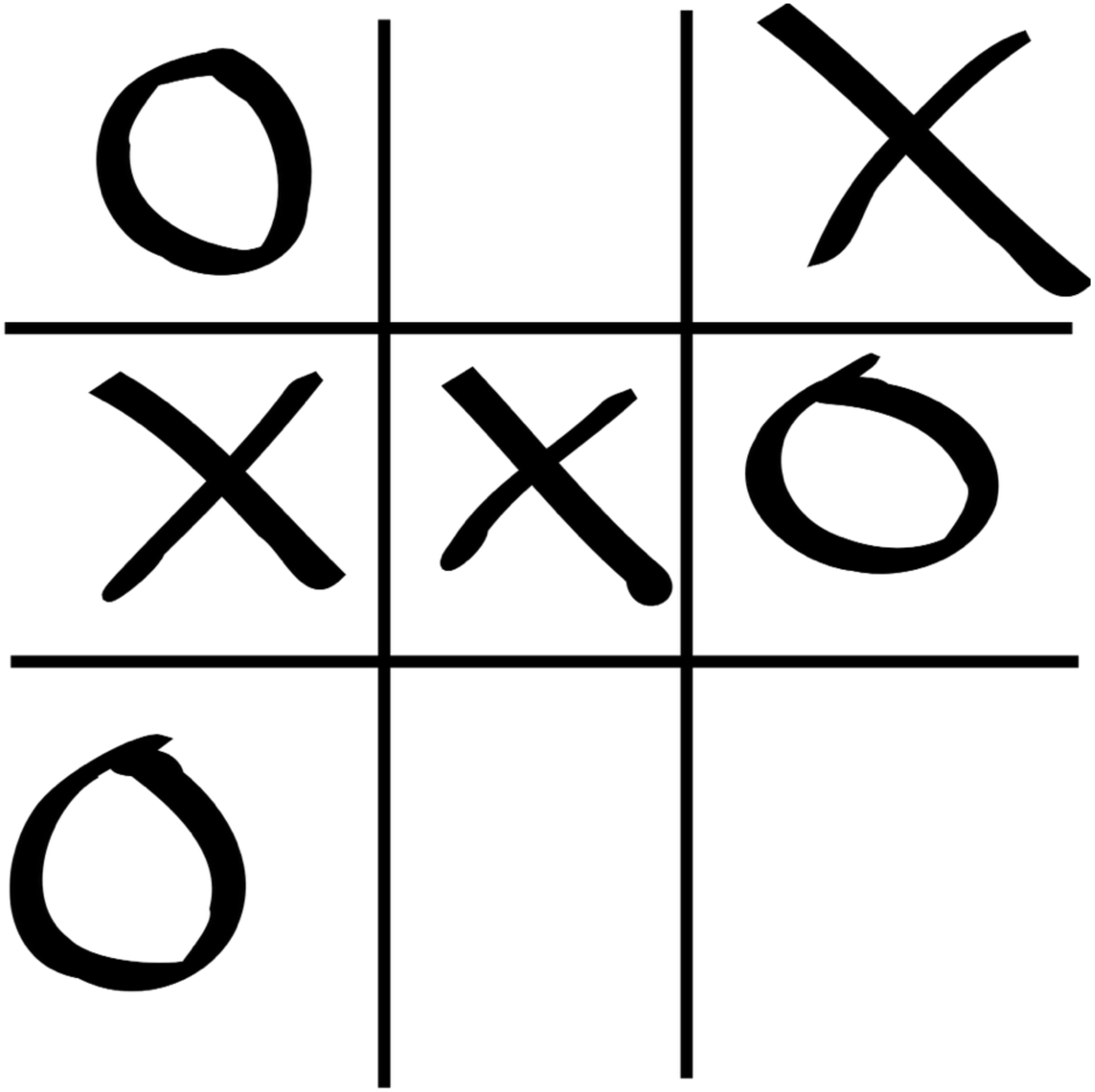
@markhibberd



“Data is validated with respect to other data

If types are to capture valid data precisely,
we must let them depend on terms”

- Conor McBride, Winging It



IDRIS
a primer

haskell like syntax

but...

types and values co-exist in the same term language

haskell

```haskell
infixr 7 :.

data List a
  = Nil
  | a :. List a

headOr :: a -> List a -> a
headOr x Nil = x
headOr _ (h :. _) = h

type String =
  List Char
```

idris

```idris
infixr 7 :.

data List a
  = Nil
  | (:.) a (List a)

headOr : a -> List a -> a
headOr x Nil = x
headOr _ (h :. _) = h

String : Type
String = List Char
```

haskell

```haskell
infixr 7 :.

data List a
  = Nil
  | a :. List a

headOr :: a -> List a -> a
headOr x Nil = x
headOr _ (h :. _) = h

type String =
  List Char
```

idris

```idris
infixr 7 :.

data List a
  = Nil
  | (:.) a (List a)

headOr : {a: Type} -> (x: a)
      -> (xs: List a) -> a
headOr {a} x Nil = x
headOr {a} _ (h :. _) = h

String : Type
String = List Char
```

haskell

```haskell
data Nat
  = Z
  | S Nat
```

idris

```idris
data Nat
  = Z
  | S Nat
```

haskell

```haskell
{-# LANGUAGE GADTs #-}

data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

idris

```idris
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```

haskell

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}

data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat
```

idris

```idris
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```

haskell

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}

data Z
data S a = S a

data Nat a :: * -> * where
  Zero :: Nat Z
  Succ :: Nat n -> Nat (S n)

data Fin n :: * -> * where
  FZero :: Nat n -> Fin (S n)
  FSucc :: Nat n -> Fin n
                 -> Fin (S n)
```

idris

```idris
data Nat : Type where
  Z : Nat
  S : Nat -> Nat

data Fin : Nat -> Type where
  fZ : Fin (S k)
  fS : Fin k -> Fin (S k)
```

haskell

```haskell
{-# LANGUAGE GADTs  #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat

data Fin :: Nat -> * where
  FZ :: Fin Z
  FS :: Fin n -> Fin (S n)
```

idris

```idris
data Nat : Type where
  Z : Nat
  S : Nat -> Nat

data Fin : Nat -> Type where
  fZ : Fin (S k)
  fS : Fin k -> Fin (S k)
```

“Stick a number in my type. What is that?”

- Manuel Chakravarty

```
1 data Vect : Nat -> Type -> Type where
2   Nil  : Vect Z a
3   (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
```

```
1 data Vect : Nat -> Type -> Type where
2   Nil  : Vect Z a
3   (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
4
5 v2 : Vect 2 String
6 v2 = "hello" :: "idris" :: Nil
```

```
1 data Vect : Nat -> Type -> Type where
2   Nil  : Vect Z a
3   (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
4
5 v2 : Vect 2 String
6 v2 = "hello" :: "idris" :: Nil
7
8 v3 : Vect ?len String
9 v3 = "tic" :: "tac" :: "type" :: Nil
```

```
 1 data Vect : Nat -> Type -> Type where
 2   Nil  : Vect Z a
 3   (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
 4
 5 v2 : Vect 2 String
 6 v2 = "hello" :: "idris" :: Nil
 7
 8 v3 : Vect ?len String
 9 v3 = "tic" :: "tac" :: "type" :: Nil
10
11 len = proof search
```

```
1 data Vect : Nat -> Type -> Type where
2   Nil  : Vect Z a
3   (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
4
5 index : Fin n -> Vect n a -> a
6 index fZ     (x::xs) = x
7 index (fS k) (x::xs) = index k xs
8 index fZ     [] impossible
```

```
1 (++) : Vect n a -> Vect m a -> Vect (n + m) a
2 (++) Nil m = m
3 (++) (h :: t) m = h :: (t ++ m)
```

```
 1 (++) : Vect n a -> Vect m a -> Vect (n + m) a
 2 (++) Nil m = m
 3 (++) (h :: t) m = h :: (t ++ m)
 4
 5 filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
 6 filter p [] = ( _ ** [] )
 7 filter p (x::xs) with (filter' p xs)
 8   | (n ** tail) =
 9     if p x then
10       ((S n) ** x::tail)
11     else
12       (n ** tail)
```

what did I just see?

dependent product types
aka $\prod$ types

dependent sum types
aka Σ types

Curry–Howard correspondence

(Intuitionistic) Logic	Programming
FALSE	$\bot$
TRUE	()
Disjunction (or)	Sum Types
Conjunction (and)	Product Types
Implication	Function Types

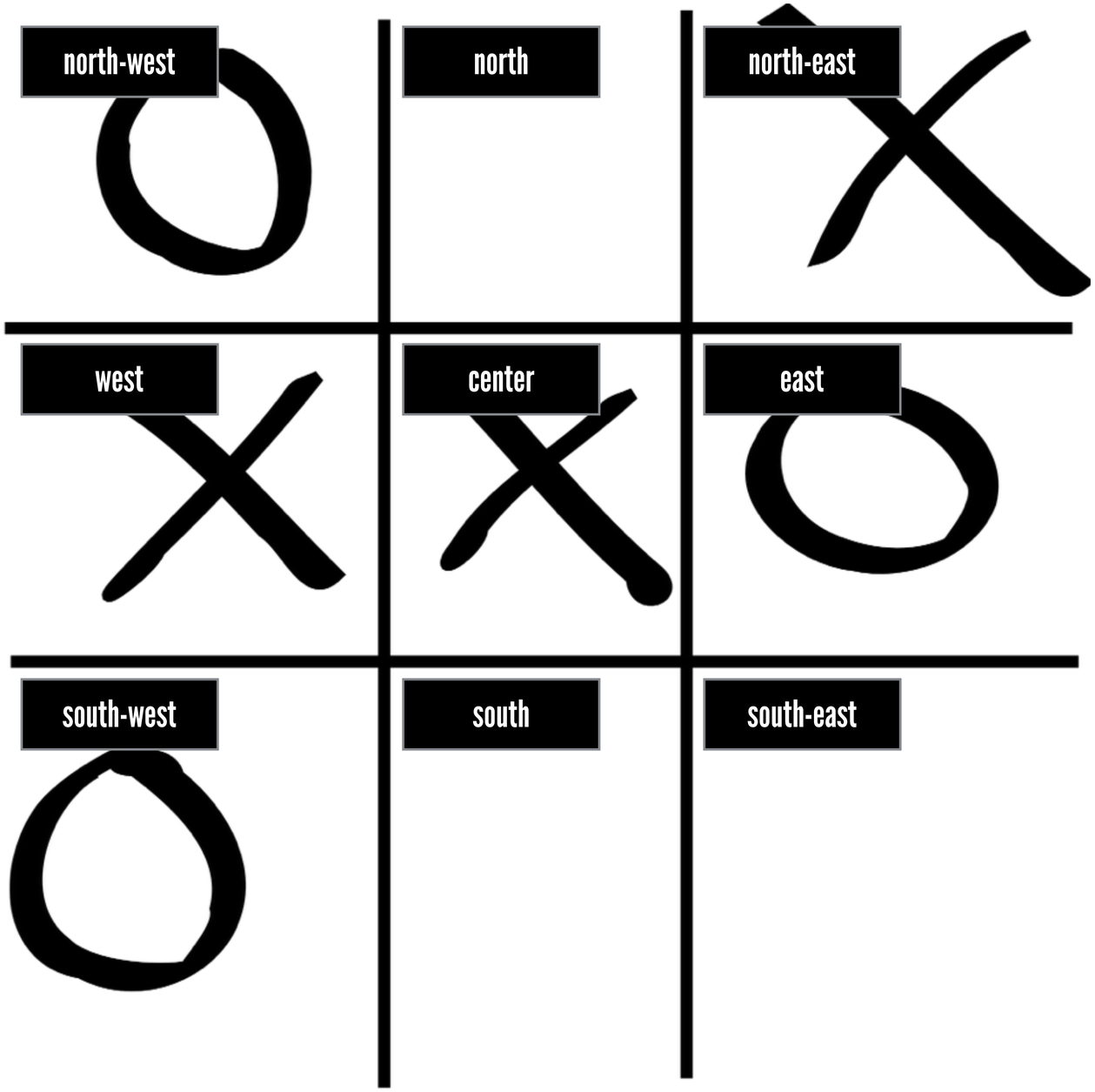
(Predicate) Logic	Programming
FALSE	$\bot$
TRUE	()
Disjunction (or)	Sum Types
Conjunction (and)	Product Types
Implication	Function Types
Existential Quantification (exists.)	Σ Types
Universal Quantification (forall.)	$\prod$ Types

So how do we go about constructing programs?

Start With Value Level Primitives

```haskell
data Player = X | O
data Cell = Occupied Player | Unoccupied

instance Show Player where {...}
instance Eq Player where {...}
instance Show Cell where {...}
instance Eq Cell where {…}
```


north-west
north
north-east
west
center
east
south-west
south
south-east

```
 1 Position : Type
 2 Position = Fin 9
 3
 4 nw : Position
 5 nw = 0
 6 {...}
 7 se : Position
 8 se = 8
 9
10 parse : String -> Maybe Position
11 parse "nw" = Just nw
12 parse "n"  = Just n
13 parse "ne" = Just ne
14 parse "e"  = Just e
15 parse "c"  = Just c
16 parse "w"  = Just w
17 parse "sw" = Just sw
18 parse "s"  = Just s
19 parse "se" = Just se
20 parse _    = Nothing
```

```
 1 data Board =
 2   B (Vect 9 Cell)
 3
 4 instance Eq Board where {...}
 5 instance Show Board where {...}
 6
 7 -- How many positions are occupied ?
 8 occupied : Board -> Nat
 9
10 -- Who's turn is it?
11 turn : Board -> Player
12
13 -- Is this position free?
14 free : Position -> Board -> Bool
15
16 -- Is there a winner on the board?
17 winner : Board -> Maybe Player
18
19 -- Is this a valid board?
20 isValidBoard : Board -> Bool
```

Types and Proofs

```
1 data so : Bool -> Type where
2   oh : so True
```

```
1 data so : Bool -> Type where
2   oh : so True
3
4 printIf3 : (n: Nat)
5   -> {default oh prf : so (n == 3)}
6   -> IO ()
7 printIf3 _ =
8   print "3s are good."
```

```
*> :x printIf3 3
"3s are good."
```

```
*> :x printIf3 3
"3s are good."

*> :x printIf3 3 { prf = oh }
"3s are good.”
```

```
*> :x printIf3 3
"3s are good."

*> :x printIf3 3 { prf = oh }
"3s are good."

*> :x printIf3 2
(input):1:13:When elaborating argument prf to
                             function Main.printIf3:
    Can't unify
            IsJust (Just x)
    with
            IsJust (fromInteger 2 == (fromInteger 3))

    Specifically:
            Can't unify
                    True
            with
                    False
```

```
1 natToFin : Nat -> (n : Nat) -> Maybe (Fin n)
2 natToFin Z     (S j) = Just fZ
3 natToFin (S k) (S j) with (natToFin k j)
4                           | Just k' = Just (fS k')
5                           | Nothing = Nothing
```

```
1 natToFin : Nat -> (n : Nat) -> Maybe (Fin n)
2 natToFin Z     (S j) = Just fZ
3 natToFin (S k) (S j) with (natToFin k j)
4                           | Just k' = Just (fS k')
5                           | Nothing = Nothing
6
7 data IsJust : Maybe a -> Type where
8   ItIsJust : IsJust {a} (Just x)
```

```
 1 natToFin : Nat -> (n : Nat) -> Maybe (Fin n)
 2 natToFin Z     (S j) = Just fZ
 3 natToFin (S k) (S j) with (natToFin k j)
 4                           | Just k' = Just (fS k')
 5                           | Nothing = Nothing
 6
 7 data IsJust : Maybe a -> Type where
 8   ItIsJust : IsJust {a} (Just x)
 9
10 fromNat : (x: Nat)
11         -> {default ItIsJust
12             prf : (IsJust (natToFin x n))}
13         -> Fin n
```

```
 1 natToFin : Nat -> (n : Nat) -> Maybe (Fin n)
 2 natToFin Z     (S j) = Just fZ
 3 natToFin (S k) (S j) with (natToFin k j)
 4                           | Just k' = Just (fS k')
 5                           | Nothing = Nothing
 6
 7 data IsJust : Maybe a -> Type where
 8   ItIsJust : IsJust {a} (Just x)
 9
10 fromNat : (x: Nat)
11         -> {default ItIsJust
12             prf : (IsJust (natToFin x n))}
13         -> Fin n
14 fromNat {n} x {prf} with (natToFin x n)
```

```
 1 natToFin : Nat -> (n : Nat) -> Maybe (Fin n)
 2 natToFin Z     (S j) = Just fZ
 3 natToFin (S k) (S j) with (natToFin k j)
 4                           | Just k' = Just (fS k')
 5                           | Nothing = Nothing
 6
 7 data IsJust : Maybe a -> Type where
 8   ItIsJust : IsJust {a} (Just x)
 9
10 fromNat : (x: Nat)
11         -> {default ItIsJust
12             prf : (IsJust (natToFin x n))}
13         -> Fin n
14 fromNat {n} x {prf} with (natToFin x n)
15   fromNat {n} x {prf = ItIsJust} | Just y = y
```

```
1 *> fromNat 7
2 Can't resolve type class Enum iType
```

```
1 *> fromNat 7
2 Can't resolve type class Enum iType
3
4 *FinProof> :t the
5 Prelude.Basics.the : (a : Type) -> a -> a
```

```
*> fromNat 7
Can't resolve type class Enum iType

*FinProof> :t the
Prelude.Basics.the : (a : Type) -> a -> a

*FinProof> the (Fin 10) (fromNat 7)
fS (fS (fS (fS (fS (fS (fS fZ))))))) : Fin 10
```

```
*> fromNat 7
Can't resolve type class Enum iType

*FinProof> :t the
Prelude.Basics.the : (a : Type) -> a -> a

*FinProof> the (Fin 10) (fromNat 7)
fS (fS (fS (fS (fS (fS (fS fZ))))))) : Fin 10

*FinProof> the (Fin 10) (fromNat 10)
(input):1:23:When elaborating argument prf
        Can't unify
                IsJust (Just x)
        with
                IsJust (natToFin 10 10)

        Specifically:
                Can't unify
                        Just x
                with
                        Nothing
```

```
1 data ValidMove : Board -> Type where
2   IsValidMove : Position -> Player
3               -> (b : Board) -> ValidMove b
```

```
1 data ValidMove : Board -> Type where
2   IsValidMove : Position -> Player
3               -> (b : Board) -> ValidMove b
4
5 tryValidMove : Position
6              -> Player
7              -> (b : Board)
8              -> Maybe (ValidMove b)
```

```
 1 data ValidMove : Board -> Type where
 2   IsValidMove : Position -> Player
 3              -> (b : Board) -> ValidMove b
 4
 5 tryValidMove : Position
 6             -> Player
 7             -> (b : Board)
 8             -> Maybe (ValidMove b)
 9
10 validMove : (pos : Position)
11           -> (p : Player)
12           -> (b : Board)
13           -> {default ItIsJust
14               prf : IsJust (tryValidMove pos p b)}
15           -> ValidMove
```

```
 1 data ValidMove : Board -> Type where
 2   IsValidMove : Position -> Player
 3              -> (b : Board) -> ValidMove b
 4
 5 tryValidMove : Position
 6             -> Player
 7             -> (b : Board)
 8             -> Maybe (ValidMove b)
 9
10 validMove : (pos : Position)
11           -> (p : Player)
12           -> (b : Board)
13           -> {default ItIsJust
14               prf : IsJust (tryValidMove pos p b)}
15           -> ValidMove
16 validMove pos p b {prf} with (tryValidMove pos p b)
```

```
 1 data ValidMove : Board -> Type where
 2   IsValidMove : Position -> Player
 3              -> (b : Board) -> ValidMove b
 4
 5 tryValidMove : Position
 6             -> Player
 7             -> (b : Board)
 8             -> Maybe (ValidMove b)
 9
10 validMove : (pos : Position)
11           -> (p : Player)
12           -> (b : Board)
13           -> {default ItIsJust
14               prf : IsJust (tryValidMove pos p b)}
15           -> ValidMove
16 validMove pos p b {prf} with (tryValidMove pos p b)
17   validMove pos p b {prf = ItIsJust} | Just y = y
```

Break on Through to the Other Side

```
1 data Game : Board -> Type where
2
3   -- start a new game with an empty board
4   start : Game empty
5
6   -- make a (guaranteed to be valid) move
7   move' : {b : Board} -> (m : ValidMove b)
8        -> Game b -> Game (runMove m)
```

```
1 -- Make a valid move
2 runMove : ValidMove board -> Board
3 runMove (IsValidMove position player (B board)) =
4   B $ replaceAt position (Occupied player) board
5
```

```
 1 -- Make a valid move
 2 runMove : ValidMove board -> Board
 3 runMove (IsValidMove position player (B board)) =
 4   B $ replaceAt position (Occupied player) board
 5
 6 -- Make a valid move if you can prove it
 7 move : {b : Board}
 8   -> (pos : Position)
 9   -> (p : Player)
10   -> (g : Game board)
11   -> {default ItIsJust
12       prf : (IsJust (tryValidMove pos p b))}
13   -> Game (runMove $ validMove pos p b {prf})
```

```
 1 -- Make a valid move
 2 runMove : ValidMove board -> Board
 3 runMove (IsValidMove position player (B board)) =
 4   B $ replaceAt position (Occupied player) board
 5
 6 -- Make a valid move if you can prove it
 7 move : {b : Board}
 8   -> (pos : Position)
 9   -> (p : Player)
10   -> (g : Game board)
11   -> {default ItIsJust
12       prf : (IsJust (tryValidMove pos p b))}
13   -> Game (runMove $ validMove pos p b {prf})
14 move {b} pos p g {prf} =
15   move' (validMove pos p b {prf}) game
```

```
1 data Prev : Game b -> Type where
2   HasPrev : {bb : Board}
3           -> {m : ValidMove bb}
4           -> {g : Game bb}
5           -> Prev (move' m g)
```

```
 1 data Prev : Game b -> Type where
 2   HasPrev : {bb : Board}
 3           -> {m : ValidMove bb}
 4           -> {g : Game bb}
 5           -> Prev (move' m g)
 6
 7 -- What was the previous board?
 8 previousBoard : (gg : Game b)
 9              -> {default HasPrev prf : (Prev gg)}
10              -> Board
```

```
data Prev : Game b -> Type where
  HasPrev : {bb : Board}
          -> {m : ValidMove bb}
          -> {g : Game bb}
          -> Prev (move' m g)

-- What was the previous board?
previousBoard : (gg : Game b)
             -> {default HasPrev prf : (Prev gg)}
             -> Board
previousBoard (move' {b} m g) {prf = HasPrev} = b
```

```
 1 data Prev : Game b -> Type where
 2   HasPrev : {bb : Board}
 3           -> {m : ValidMove bb}
 4           -> {g : Game bb}
 5           -> Prev (move' m g)
 6
 7 -- What was the previous board?
 8 previousBoard : (gg : Game b)
 9              -> {default HasPrev prf : (Prev gg)}
10              -> Board
11 previousBoard (move' {b} m g) {prf = HasPrev} = b
12
13 -- Take back the last move?
14 takeBack : (gg : Game b)
15          -> {default HasPrev prf : (Prev gg)}
16          -> Game (previousBoard gg {prf})
```

```
 1 data Prev : Game b -> Type where
 2   HasPrev : {bb : Board}
 3           -> {m : ValidMove bb}
 4           -> {g : Game bb}
 5           -> Prev (move' m g)
 6
 7 -- What was the previous board?
 8 previousBoard : (gg : Game b)
 9              -> {default HasPrev prf : (Prev gg)}
10              -> Board
11 previousBoard (move' {b} m g) {prf = HasPrev} = b
12
13 -- Take back the last move?
14 takeBack : (gg : Game b)
15          -> {default HasPrev prf : (Prev gg)}
16          -> Game (previousBoard gg {prf})
17 takeBack (move' {b} m g) {prf = HasPrev} = g
```

```
1 game0 : ?t1
2 game0 = start
3 t1 = proof search
```

```
1 game0 : ?t1
2 game0 = start
3 t1 = proof search
4
5 --game0x : ?t2
6 --game0x = takeBack game0
7 --t3 = proof search
8
```

```
 1 game0 : ?t1
 2 game0 = start
 3 t1 = proof search
 4
 5 --game0x : ?t2
 6 --game0x = takeBack game0
 7 --t3 = proof search
 8
 9 game1 : ?t3
10 game1 = move ne X game0
11 t3 = proof search
```

```
 1 game0 : ?t1
 2 game0 = start
 3 t1 = proof search
 4
 5 --game0x : ?t2
 6 --game0x = takeBack game0
 7 --t3 = proof search
 8
 9 game1 : ?t3
10 game1 = move ne X game0
11 t3 = proof search
12
13 --game1x : ?t4
14 --game1x = move ne O game1
15 --t4 = proof search
```

```
 1 game0 : ?t1
 2 game0 = start
 3 t1 = proof search
 4
 5 --game0x : ?t2
 6 --game0x = takeBack game0
 7 --t3 = proof search
 8
 9 game1 : ?t3
10 game1 = move ne X game0
11 t3 = proof search
12
13 --game1x : ?t4
14 --game1x = move ne O game1
15 --t4 = proof search
16
17 game2 : ?t5
18 game2 = move sw O game1
19 t5 = proof search
```

Existentials for Free

```
 1 data TicTacToeState =
 2   InPlay | Done
 3
 4 toState : Board -> TicTacToeState
 5 toState b =
 6    case (winner b, complete b) of
 7      (Just p, _)  => Done
 8      (Nothing, True)  => Done
 9      (Nothing, False) => InPlay
10
11 data TicTacToe : TicTacToeState -> Type where
12   T : (Game b) -> TicTacToe (toState b)
```

Effects

```
Eff : (m : Type -> Type)
   -> (x : Type)
   -> List EFFECT
   -> (x -> List EFFECT)
   -> Type
```

```
 1 Eff : (m : Type -> Type)
 2    -> (x : Type)
 3    -> List EFFECT
 4    -> (x -> List EFFECT)
 5    -> Type
 6
 7
 8 STATE : Type -> EFFECT
 9 EXCEPTION : Type -> EFFECT
10 FILEIO : Type -> EFFECT
11 STDIO : EFFECT
12 RND : EFFECT
```

```
 1 data TicTacToeRules : Effect where
 2   Move :
 3          (position : Position)
 4       -> (player : Player)
 5       -> { TicTacToe st  ==> {st'} (TicTacToe st')  }
 6          TicTacToeRules TicTacToeState
 7
 8  Get : { g } TicTacToeRules g
 9
10  GetBoard : { TicTacToe st } TicTacToeRules Board
11
```

```
 1 data TicTacToeRules : Effect where
 2   Move :
 3          (position : Position)
 4       -> (player : Player)
 5       -> { TicTacToe st  ==> {st'} (TicTacToe st')  }
 6          TicTacToeRules TicTacToeState
 7
 8  Get : { g } TicTacToeRules g
 9
10  GetBoard : { TicTacToe st } TicTacToeRules Board
11
12 TICTACTOE : TicTacToeState -> EFFECT
13 TICTACTOE s = MkEff (TicTacToe s) TicTacToeRules
```

```
1 using (m : Type -> Type)
2  instance Handler TicTacToeRules m where
3   handle (T game) (Move position player) k
4   handle t Get k
5   handle (T game) GetBoard k
```

```
using (m : Type -> Type)
 instance Handler TicTacToeRules m where
  handle (T game) (Move position player) k =
    let b = toBoard game in
    case tryValidMove position player b of
     Just m' =>
       let updated = move' m' game in
       k (toState . toBoard $ updated) (T updated)
     Nothing =>
       k (toState . toBoard $ game) (T game)

  handle t Get k =
    k t t

  handle (T game) GetBoard k =
    k (toBoard game) (T game)
```

```
1 game : { [TICTACTOE InPlay, STDIO] ==>
2          [TICTACTOE Done, STDIO] } Eff IO ()
3 game = do
```

```
 1 game : { [TICTACTOE InPlay, STDIO] ==>
 2           [TICTACTOE Done, STDIO] } Eff IO ()
 3 game = do
 4   let current = !GetBoard
 5   let player = turn current
 6   let _ = !(printState current player)
 7   let input = !getStr
 8   case parse (trim input) of
 9     Nothing => '
10       do putStrLn $ "Invalid move:  " ++ input
11          pure !game
12     Just position =>
13       do InPlay <- Move position player
14             | Done => putStrLn "Done"
15          game
```

Constraints are Context Dependent

```
1 data CurrentGame : Type where
2   Current : (TicTacToe state) -> CurrentGame
```

```
1 data CurrentGame : Type where
2   Current : (TicTacToe state) -> CurrentGame
3
4 receive' : { [STDIO,
5               STATE CurrentGame,
6               TCPSERVERCLIENT ClientConnected] ==>
7              [STDIO,
8               STATE CurrentGame,
9               TCPSERVERCLIENT ()] } Eff IO ()
```

```
 1 data CurrentGame : Type where
 2   Current : (TicTacToe state) -> CurrentGame
 3
 4 receive' : { [STDIO,
 5               STATE CurrentGame,
 6               TCPSERVERCLIENT ClientConnected] ==>
 7              [STDIO,
 8               STATE CurrentGame,
 9               TCPSERVERCLIENT ()] } Eff IO ()
10
11 game : CurrentGame -> Game x
```

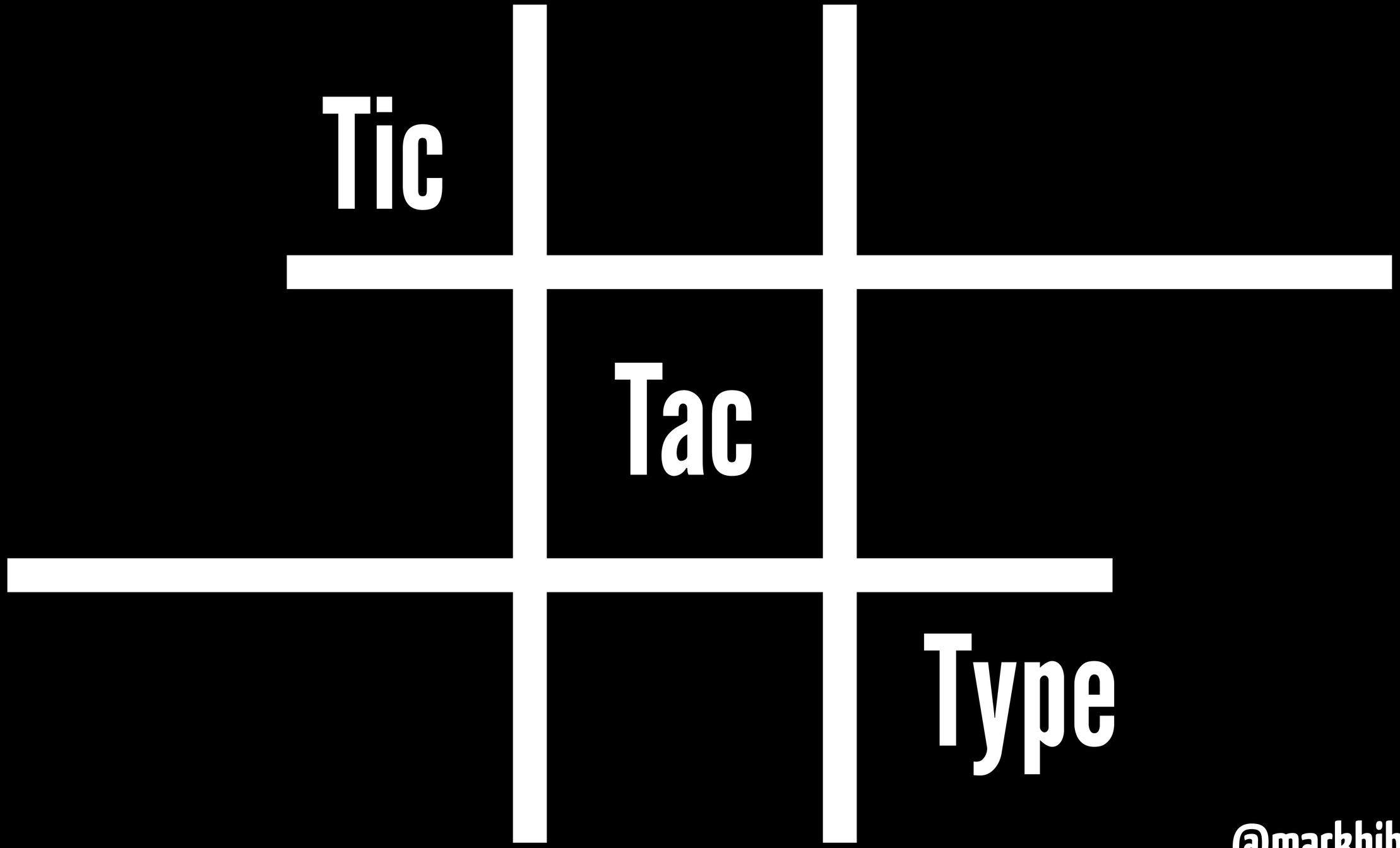
```
 1 data CurrentGame : Type where
 2   Current : (TicTacToe state) -> CurrentGame
 3
 4 receive' : { [STDIO,
 5               STATE CurrentGame,
 6               TCPSERVERCLIENT ClientConnected] ==>
 7              [STDIO,
 8               STATE CurrentGame,
 9               TCPSERVERCLIENT ()] } Eff IO ()
10
11 game : CurrentGame -> Game x –??????????
```

```
 1 data CurrentGame : Type where
 2   Current : (TicTacToe state) -> CurrentGame
 3
 4 receive' : { [STDIO,
 5               STATE CurrentGame,
 6               TCPSERVERCLIENT ClientConnected] ==>
 7              [STDIO,
 8               STATE CurrentGame,
 9               TCPSERVERCLIENT ()] } Eff IO ()
10
11 game : CurrentGame -> (x ** Game x)
12 game (Current (T z)) =
13   (toBoard z ** z)
```

```
 1 data CurrentGame : Type where
 2   Current : (TicTacToe state) -> CurrentGame
 3
 4 receive' : { [STDIO,
 5               STATE CurrentGame,
 6               TCPSERVERCLIENT ClientConnected] ==>
 7              [STDIO,
 8               STATE CurrentGame,
 9               TCPSERVERCLIENT ()] } Eff IO ()
10
11 game : CurrentGame -> (x ** Game x)
```

“the most popular and best established lightweight formal methods are *type systems*”

- Benjamin Pierce, Types and Programming Languages



@markhibberd