$$\Gamma \models \Delta$$

We can use the rules to show this is universally valid,
or, if it is not, to generate
a counterexample, a model in which

$$\Gamma \not\models \Delta$$

$$\text{some } \bigwedge \Gamma \text{ is not } \bigvee \Delta$$

Can we use the rules to show this is somewhere valid?
We say the sequent is **satisfiable** if we can
find a model in which

$$\text{some } \bigwedge \Gamma \text{ is } \bigvee \Delta$$

Can we use the rules to show this is somewhere valid?
We say the sequent is satisfiable if we can
find a model where

$$\text{some } \bigwedge \Gamma \text{ is } \bigvee \Delta$$

$$\Gamma \not\models \neg \bigvee \Delta$$

$$\Gamma \models \neg \bigvee \Delta$$

We can use the rules to show this is universally valid,
or, if it is not, to generate
a counterexample, which shows

$$\Gamma \not\models \neg \bigvee \Delta$$

$\Gamma \models \neg \bigvee \Delta$ $\quad \Gamma, \bigvee \Delta \models$

We can use the rules to show this is universally valid,

$\Gamma, \bigvee \Delta$ is inconsistent

or, if it is not, to generate
a counterexample, a model in which

$$\Gamma \not\models \neg \bigvee \Delta$$

$$\text{some } \bigwedge \Gamma \text{ is } \bigvee \Delta$$

$$\models$$

$$\bigwedge \varnothing \models \bigvee \varnothing$$

$$\top \models \bot$$

which is only valid in the empty universe

$$\vDash$$

$$\Gamma \vDash \Delta \quad (\Gamma = \Delta = \varnothing)$$

$$\bigwedge \varnothing \vDash \bigvee \varnothing$$

$$\top \vDash \bot$$

which is only valid in the empty universe

$$\varnothing \vDash \varnothing$$

$$a \vDash b \quad (a = b = \varnothing = \bot)$$

$$\bot \vDash \bot$$

which is universally true

This is a type error
— but for a mathematician
a set is just a set
there is only one emptyset

Haskell keeps track of what we are talking about
— and tells us when we are talking nonsense

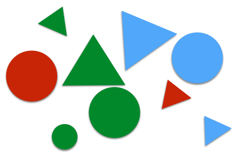
```
Prelude> 1 : [] :: [Int]
[1]
Prelude> tail it
[]
Prelude> False : it

<interactive>:26:9: error:
    •Couldn't match type 'Int' with 'Bool'
```

```
(&&) :: Bool -> Bool -> Bool

a       :: U -> Bool
b       :: U -> Bool
a &:& b :: U -> Bool

(&:&)   :: (u -> Bool) -> (u -> Bool) -> u -> Bool
(&:&) a b x = a x && b x

a       :: U -> Bool
b       :: U -> Bool
a &:& b :: U -> Bool

(&:&) :: (u -> Bool) -> (u -> Bool) -> (u -> Bool)
(a &:& b) x = a x && b x
```

```
a       :: U -> Bool
b       :: U -> Bool
a &:& b :: U -> Bool

(&:&) :: (u -> Bool) -> (u -> Bool) -> (u -> Bool)
(a &:& b) x = a x && b x

type Pred u = u -> Bool
a       :: Pred u
b       :: Pred u
a &:& b :: Pred u

(&:&) :: Pred u -> Pred u -> Pred u
(a &:& b) x = a x && b x
```

```
data Bool = False | True
not  :: Bool -> Bool
(&&) :: Bool -> Bool -> Bool -- ∧
(||) :: Bool -> Bool -> Bool -- ∨
(<=) :: Bool -> Bool -> Bool -- →
(==) :: Bool -> Bool -> Bool -- ↔
(/=) :: Bool -> Bool -> Bool -- ⊕
 and :: [ Bool] -> Bool      -- ⋀
 or  :: [ Bool] -> Bool      -- ⋁
 -- predicates are functions defined on some universe
 -- (normally finite) operations on predicates are defined
 --  by 'lifting' operations operations on Bool
TT     :: a -> Bool
FF     :: a -> Bool
neg    :: (a -> Bool) -> (a -> Bool)
(:&:)  :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
(:|:)  :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
bigand :: [Pred a] -> Pred a
bigor  :: [Pred a] -> Pred a
```

```haskell
data Bool = False | True
not  :: Bool -> Bool
(&&) :: Bool -> Bool -> Bool -- ∧
(||) :: Bool -> Bool -> Bool -- ∨
(<=) :: Bool -> Bool -> Bool -- →
(==) :: Bool -> Bool -> Bool -- ↔
(/=) :: Bool -> Bool -> Bool -- ⊕
 and :: [ Bool] -> Bool      -- ⋀
 or  :: [ Bool] -> Bool      -- ⋁
 -- predicates are functions defined on some universe
 -- (normally finite) operations on predicates are defined
 --  by 'lifting' operations operations on Bool
type Pred a = a -> Bool
TT     :: Pred a
FF     :: Pred a
neg    :: Pred a -> Pred a
(:&:)  :: Pred a -> Pred a -> Pred a
(:|:)  :: Pred a -> Pred a -> Pred a
bigand :: [Pred a] -> Pred a
bigor  :: [Pred a] -> Pred a
```



every small triangle is red

```haskell
and [ isRed x | x <- things, isSmall x, isTriangle x ]
```

some small triangle is red

```haskell
or [ isRed x | x <- things, isSmall x, isTriangle x ]
```

every small triangle is red

```haskell
and [ isRed x | x <- things, isSmall x, isTriangle x ]
```

some small triangle is red

```haskell
or [ isRed x | x <- things, isSmall x, isTriangle x ]
```

every small triangle is red

```
and [ isRed x | x <- things, isSmall x, isTriangle x ]
```

some small triangle is red

```
or [ isRed x | x <- things, isSmall x, isTriangle x ]
```

every small triangle is red

```
and [ isRed x | x <- things, (isSmall &:& isTriangle) x ]
```

some small triangle is red

```
or [ isRed x | x <- things, (isSmall &:& isTriangle) x ]
```

```
isHappy :: Person -> Bool
```

everybody is happy

```
body :: [Person]
and [ isHappy x | x <- body ]

every xs p = and [ p x | x <- xs ]
every :: [t] -> (t -> Bool) -> Bool
every body isHappy
```

```
every :: [t] -> (t -> Bool) -> Bool
every xs p = and [ p x | x <- xs ]

loves   :: Person -> Person -> Bool
body = [Krithik,Kristin,Callum,Muhammad,Sapphira,
        Jessica,Gabrielle,Katie,Divy,Mary,Mark,...]

loves Mark Mary
Mark `loves` Mary
loves Mark :: ????
```

```
every :: [t] -> (t -> Bool) -> Bool
every xs p = and [ p x | x <- xs ]

loves   :: Person -> Person -> Bool
body = [Krithik,Kristin,Callum,Muhammad,Sapphira,
        Jessica,Gabrielle,Katie,Divy,Mary,Mark,...]

loves Mark Mary
Mark `loves` Mary
loves Mark :: Person -> Bool
```

what does this mean ?

```
every body (loves Mark)
```

```
every :: [t] -> (t -> Bool) -> Bool
every xs p = and [ p x | x <- xs ]
loves Mark Mary
Mark `loves` Mary

every body (loves Mark)
  = and [ loves Mark x | x <- body ]
  = and [  Mark `loves` x | x <- body ]
```

Mark loves every body !

Mark loves every body !

`loves Mark` -- really means *Mark loves*

Haskell knows this!

```
(Mark `loves`) :: Person -> Bool
(Mark `loves`)  x = Mark `loves` x
                  = loves Mark x
```

```
every :: [t] -> (t -> Bool) -> Bool
every xs p = and [ p x | x <- xs ]
loves Mark Mary
Mark `loves` Mary

every body (loves Mark)
  = and [ loves Mark x | x <- body ]
  = and [ Mark `loves` x | x <- body ]
  = and [ (Mark `loves`) x | x <- body ]
```

Mark loves every body !

```
some :: [t] -> (t -> Bool) -> Bool
some xs p = or [ p x | x <- xs ]
Mark `loves` Mary
```

some body loves Mary

```
or [ b `loves` Mary | b <- body ]

lovesMary :: Person -> Bool
lovesMary x = x `loves` Mary
some body lovesMary
some body (`loves` Mary)
```

Sections

```
(`loves` Mary) x = x `loves` Mary
(Mark `loves`) y = Mark `loves` y
```

somebody loves everybody
everybody loves somebody

```
every body (Mary `loves`)                 -- Mary loves everybody
lovesEveryBody x = every body (x `loves`) -- x loves everybody
someBodyLovesEveryBody = some body lovesEveryBody
```

λ *lambda*

```
square x = x * x
square = (\x -> x * x) -- λx.x × x
hypotenuse a b = sqrt (square a + square b)
hypotenuse = (\a b -> sqrt (square a + square b))
           -- λ a b . √(x² + y²)
```

```
(`loves` Mary) x = x `loves` Mary
(`loves` Mary) = (\x -> x `loves` Mary)
some body (`loves` Mary) = some body (\x -> x `loves` Mary)

(Mark `loves`) y = Mark `loves` y
(Mark `loves`) = (\y -> Mark `loves` y)
every body (Mark `loves`) = every body (\y -> Mark `loves` y)
```

everybody loves somebody

```
EveryBodyLovesSomeBody = every body (\x -> some body (\y -> x `loves` y))
example2 = some body (\x -> every body (\y -> x `loves` y)) -- ??
example3 = some body (\x -> every body (\y -> y `loves` x)) -- ??
example3 = every body (\x -> some body (\y -> y `loves` x)) -- ??
```

Sections

`(> 0)` is shorthand for `(\x -> x > 0)`

`(2 *)` is shorthand for `(\x -> 2 * x)`

`(+ 1)` is shorthand for `(\x -> x + 1)`

`(2 ^)` is shorthand for `(\x -> 2 ^ x)`

`(^ 2)` is shorthand for `(\x -> x ^ 2)`

```
(&:&) :: (u -> Bool) -> (u -> Bool) -> (u -> Bool)
a &:& b = (\x -> a x && b x)
```

```
data Literal a = P a | N a
data Clause a  = Or[ Literal a ]
type Form a    = [ Clause a ]

neg :: Literal a -> Literal a
neg (P a) = N a
neg (N a) = P a

data Atom = A|B|C|D|W|X|Y|Z deriving Eq

eg = [ Or[N A, N C, P D], Or[P A, P C], Or[N D] ]

data Val a = And [ Literal a ]
```