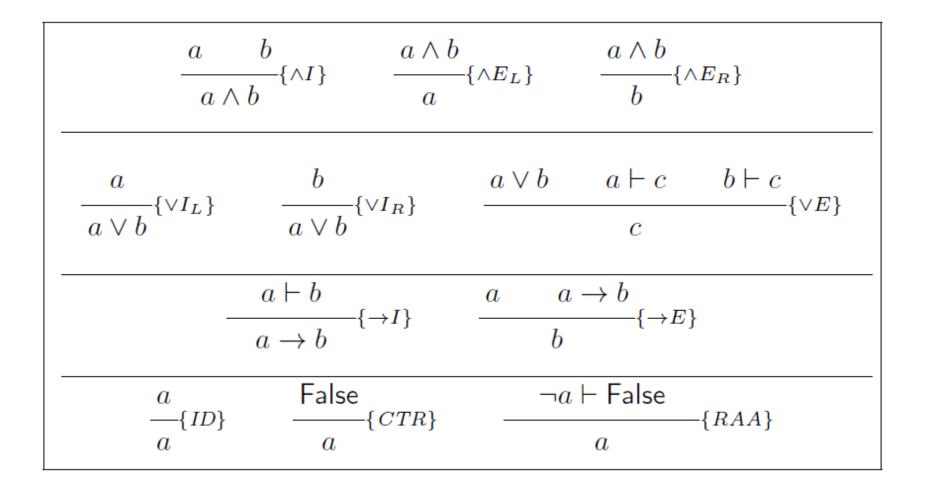
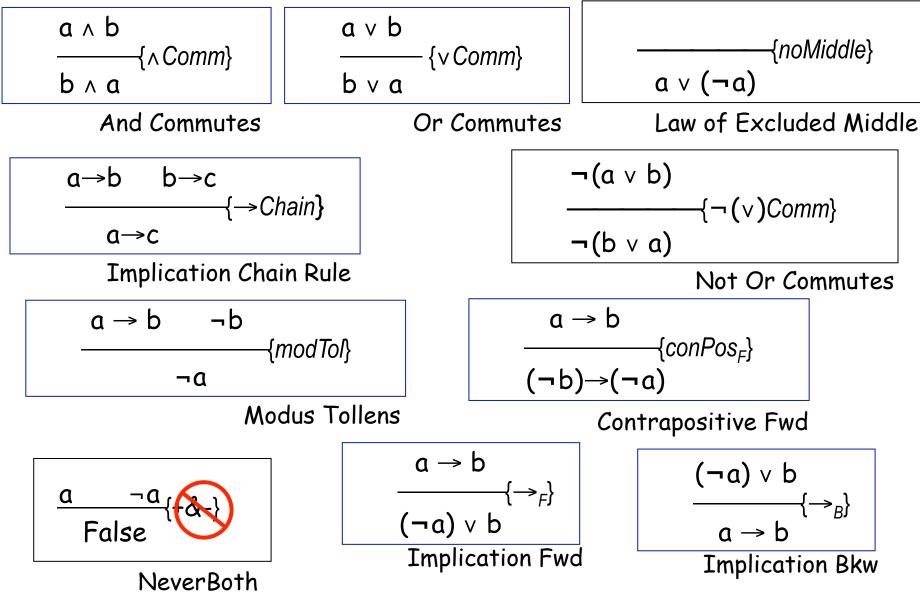
#### Inference Rules: Propositional Calculus

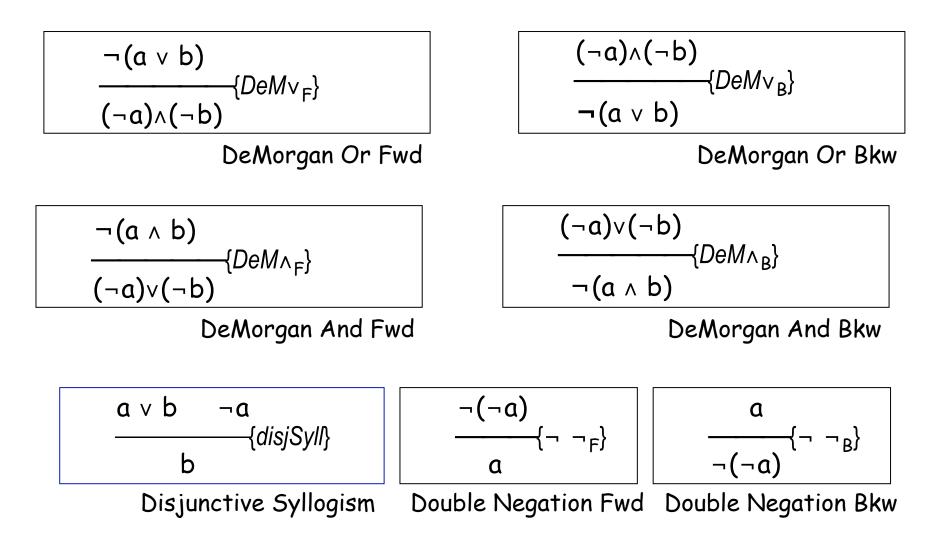


1

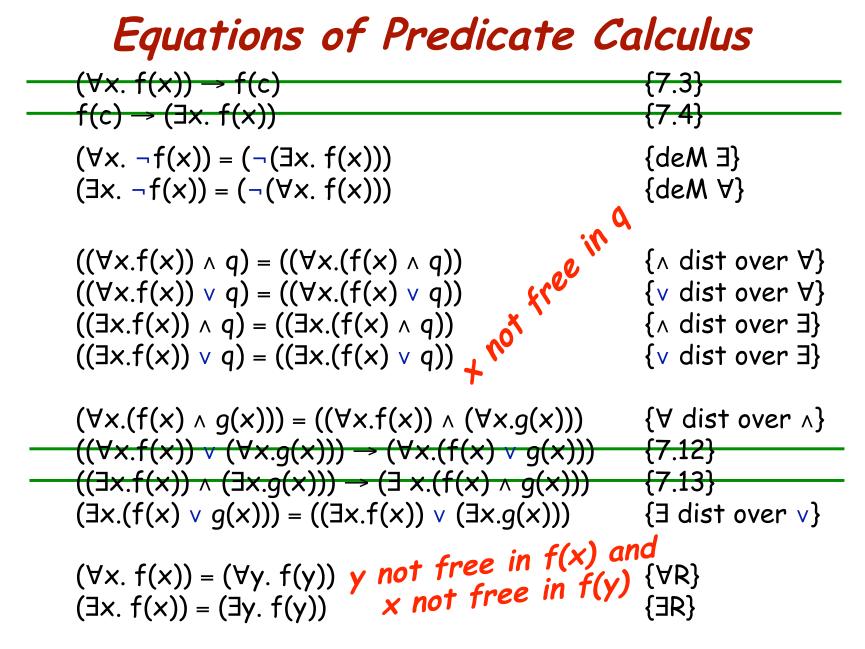
### Some Theorems in Rule Form



#### More Theorems in Rule Form

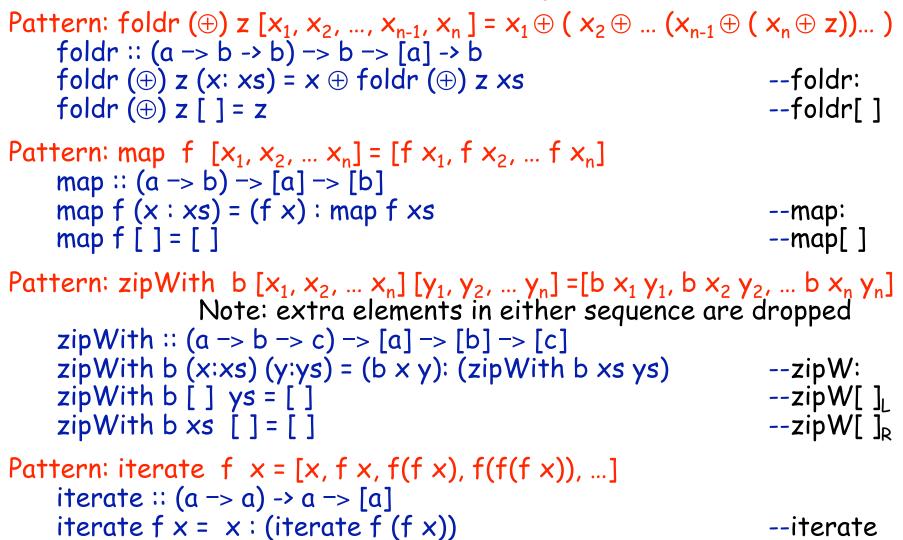


```
a \wedge False = False
                                                 \{ \land null \}
a \vee True = True
                                                  \{ \vee null \}
a \wedge True = a
                                                  \{\land identity\}
                                                                           Some Equations of
a \vee False = a
                                                  \{\vee identity\}
                                                  \{\land idempotent\}
a \wedge a = a
                                                                               Boolean Algebra
                                                  {v idempotent}
a \vee a = a
a \wedge b = b \wedge a
                                                  \{\land commutative\}
a \vee b = b \vee a
                                                  \{ \lor commutative \}
(a \land b) \land c = a \land (b \land c)
                                                  {^ associative}
(a \lor b) \lor c = a \lor (b \lor c)
                                                 {v associative}
\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c}) \{ \wedge \text{ distributes over } \vee \}
\mathbf{a} \lor (\mathbf{b} \land \mathbf{c}) = (\mathbf{a} \lor \mathbf{b}) \land (\mathbf{a} \lor \mathbf{c}) \{\lor \text{ distributes over } \land\}
\neg (a \land b) = (\neg a) \lor (\neg b)
                                                 {DeMorgan's law \land}
\neg (a \lor b) = (\neg a) \land (\neg b)
                                                 {DeMorgan's law \vee}
\negTrue = False
                                                  {negate True}
\neg False = True
                                                 {negate False}
(a \land (\neg a)) = False
                                                  \{\land complement\}
(a \lor (\neg a)) = True
                                                  \{ \lor complement \}
\neg(\neg a) = a
                                                 {double negation}
(a \land b) \rightarrow c = a \rightarrow (b \rightarrow c)
                                                  {Currying}
a \rightarrow b = (\neg a) \lor b
                                                  {implication}
                                                                                       Theorems
a \rightarrow b = (\neg b) \rightarrow (\neg a)
                                                 {contrapositive}
                                                              (a \land b) \lor b = b
                                                                                                                    {v absorption}
            Axioms
                                                               (a \lor b) \land b = b
                                                                                                                    \{\land absorption\}
                                                              (a \lor b) \rightarrow c = (a \rightarrow c) \land (b \rightarrow c) \{\lor imp\}
                                                                                                                                      4
```



#### Inductive Equations (axioms) and Some Theorems sum :: Num n => [n] -> n sum(x: xs) = x + sum xssum: sum[] = 0 sum[] Theorem: sum = foldr (+) 0 sum.foldr length :: [a] -> Int length(x: xs) = 1 + length xs length.: length[] = 0length.[] Theorem: length = foldr oneMore 0 length.foldr (++) :: [a] -> [a] -> [a] (x: xs) ++ ys = x: (xs ++ ys) ++ : [ ] ++ ys = ys ++[] Theorem: xs ++ ys = foldr (:) ys xs ++.foldr Theorem: length(xs ++ ys) = (length xs) + (length ys) ++.additive Theorem: ((xs ++ ys) ++ zs) = (xs ++ (ys ++ zs))++.assoc concat :: [[a]] -> [a] concat(xs: xss) = xs ++ concat xss concat: concat[] = [] concat.[] Theorem: concat = foldr (++) [] concat.foldr (x: []) = [x] :[] $(xs \neq []) = (3x. 3ys. (xs = (x: ys)))$ (:) (: ...) $(x : [x_1, x_2, ...]) = [x, x_1, x_2, ...]$ 6

### Patterns of Computation



## Predicates, Quantifiers, and Variables

Predicate - parameterized collection of propositions

- P(x) is a proposition from predicate P
- x comes from the universe of discourse, which must be specific

 $\Box \forall x.P(x) - \forall$  quantifier converts predicate to proposition

False if and only if there is some x for which P(x) is False

 $\Box$   $\exists x.P(x) - \forall$  quantifier converts predicate to proposition

True if and only if there is some x for which P(x) is True

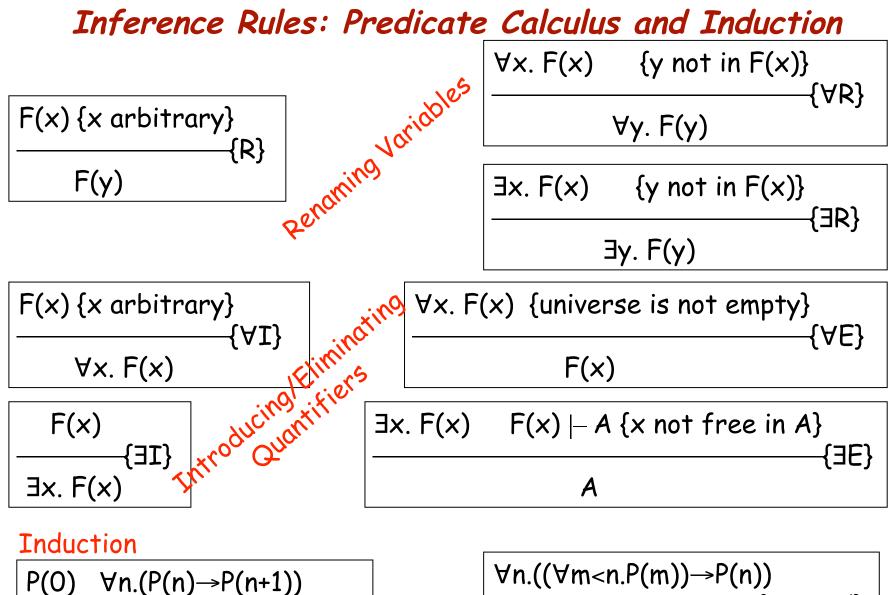
□ Free and bound variables in predicate calculus formulas

#### Bound variable

- $\checkmark \forall x. e$  x is bound in the formula  $\forall x. e$
- $\checkmark$  3x. e x is bound in the formula 3x. e
- Free variables are variables that are not bound

Arbitrary variables in proofs

 A free variable in a predicate calculus formula is <u>arbitrary</u> in a proof if it does not occur free in any undischarged assumption of that proof



## Principle of Mathematical Induction another way to skin a cat

- □{∀I} an inference rule with ∀n. P(n) as it's conclusion
   □One way to use {∀I}
  - Prove P(0)
  - Prove P(n +1) for arbitrary n
    ✓ Takes care of P(1), P(2), P(3), ...

$$\begin{array}{|c|c|} \hline P(0) & \forall n.P(n) \rightarrow P(n+1) \\ \hline & & \\ \hline & & \\ \hline & \forall n. P(n) \end{array} \end{array}$$

Induction

Mathematical induction makes it easier

Proof of P(n +1) can cite P(n) as a reason

✓ If you cite P(n) as a reason in proof of P(n+1), your proof relies on mathematical induction

- ✓ If you don't, your proof relies on {∀I}
- Strong induction makes it even easier

 $\checkmark$  The proof of P(n+1) can cite P(n), P(n-1), ... and/or P(0)

# Haskell Type Specifications

- $\Box$  x, y, z :: Integer
- □ xs, ys :: [Integer]
- □ xy :: (Integer, Bool)
- $\Box$  or :: [Bool] -> Bool
- □ (++) :: [e] -> [e] -> [e]

 $\Box$  sum :: Num n => [n] -> n

- -- x, y, and z have type Integer
- -- sequences with Integer elements
- -- 2-tuple with 1<sup>st</sup> component Integer, 2<sup>nd</sup> Bool
- -- function with one argument argument is sequence with Bool elems delivers value of type Bool
- -- generic function with two arguments args are sequences with elems of same type type is not constrained (can be any type) delivers sequence with elements of same type as those in arguments
- -- generic function with one argument argument is a sequence with elems of type n n must a type of class Num Num is a set of types with +, \*, ... operations

powerSet :: (Eq e, Show e) => Set e -> Set(Set e)

-- generic function with one argument argument is a set with elements of type e delivers set with elements of type (Set e) type e must be both class Eq and class Show Class Eq has == operator, Show displayable