Some Equations of Boolean Algebra

Axioms

$a \wedge False = False$	{∧ *null*}
$a \vee True = True$	{∨ *null*}
$a \wedge True = a$	{∧ *identity*}
$a \vee False = a$	{∨ *identity*}
$a \wedge a = a$	{∧ *idempotent*}
$a \vee a = a$	{∨ *idempotent*}
$a \wedge b = b \wedge a$	{∧ *commutative*}
$a \vee b = b \vee a$	{∨ *commutative*}
$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	{∧ *associative*}
$(a \vee b) \vee c = a \vee (b \vee c)$	{∨ *associative*}
$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	{∧ *distributes over* ∨}
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	{∨ *distributes over* ∧}
$\neg(a \wedge b) = (\neg a) \vee (\neg b)$	{*DeMorgan's law* ∧}
$\neg(a \vee b) = (\neg a) \wedge (\neg b)$	{*DeMorgan's law* ∨}
$\neg True = False$	{*negate True*}
$\neg False = True$	{*negate False*}
$(a \wedge (\neg a)) = False$	{∧ *complement*}
$(a \vee (\neg a)) = True$	{∨ *complement*}
$\neg(\neg a) = a$	{*double negation*}
$(a \wedge b) \to c = a \to (b \to c)$	{*Currying*}
$a \to b = (\neg a) \vee b$	{*implication*}
$a \to b = (\neg b) \to (\neg a)$	{*contrapositive*}

Theorems

$(a \wedge b) \vee b = b$	{∨ *absorption*}
$(a \vee b) \wedge b = b$	{∧ *absorption*}
$(a \vee b) \to c = (a \to c) \wedge (b \to c)$	{∨ *imp*}

Equations of Predicate Calculus

~~$(\forall x.\ f(x)) \to f(c)$~~	~~{7.3}~~
~~$f(c) \to (\exists x.\ f(x))$~~	~~{7.4}~~
$(\forall x.\ \neg f(x)) = (\neg(\exists x.\ f(x)))$	{deM ∃}
$(\exists x.\ \neg f(x)) = (\neg(\forall x.\ f(x)))$	{deM ∀}
$((\forall x.f(x)) \wedge q) = ((\forall x.(f(x) \wedge q))$	{∧ dist over ∀}
$((\forall x.f(x)) \vee q) = ((\forall x.(f(x) \vee q))$	{∨ dist over ∀}
$((\exists x.f(x)) \wedge q) = ((\exists x.(f(x) \wedge q))$	{∧ dist over ∃}
$((\exists x.f(x)) \vee q) = ((\exists x.(f(x) \vee q))$	{∨ dist over ∃}

x not free in q

$(\forall x.(f(x) \wedge g(x))) = ((\forall x.f(x)) \wedge (\forall x.g(x)))$	{∀ dist over ∧}
~~$((\forall x.f(x)) \vee (\forall x.g(x))) \to (\forall x.(f(x) \vee g(x)))$~~	~~{7.12}~~
~~$(\exists x.(f(x) \wedge g(x))) \to ((\exists x.f(x)) \wedge (\exists x.g(x)))$~~	~~{7.13}~~
$(\exists x.(f(x) \vee g(x))) = ((\exists x.f(x)) \vee (\exists x.g(x)))$	{∃ dist over ∨}
$(\forall x.\ f(x)) = (\forall y.\ f(y))$	{∀R}
$(\exists x.\ f(x)) = (\exists y.\ f(y))$	{∃R}

y not free in f(x) and x not free in f(y)

2

Some Software Axioms

- Axiom of sequence construction
  x : [x_1, x_2, …, x_n] = [x, x_1, x_2, …, x_n] -- (:…)
- Axioms of concatenation
  [ ] ++ ys = ys -- (++)[ ]
  (x : xs) ++ ys = x : (xs ++ ys) -- (++):
  What is the type of (++) ?
  (++) :: [a] -> [a] -> [a]
- Axioms of foldr
  foldr (⊕) z [ ] = z -- foldr[ ]
  foldr (⊕) z (x : xs) = x ⊕ (foldr (⊕) z xs) -- foldr:
  What is the type of foldr ?
  foldr :: (a -> b -> b) -> b -> [a] -> b
- The "big or" axiom
  (\/) :: Bool -> Bool -> Bool -- "little or" – satisfies Boolean laws for ∨
  or = foldr (\/) False -- "big or"
  What is the type of or ?
  or :: [Bool] -> Bool

3

Inductive Equations (axioms) and Some Theorems

sum :: Num n => [n] -> n	
sum(x: xs) = x + sum xs	sum :
sum[ ] = 0	sum[ ]
Theorem: sum = foldr (+) 0	sum.foldr
length :: [a] -> Int	
length(x: xs) = 1 + length xs	length.:
length[ ] = 0	length.[ ]
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 ≠ [ ]) = (∃x. ∃ys. (xs = (x: ys)))	(:)
(x : [x_1, x_2, …]) = [x, x_1, x_2, …]	(: …)

4

Patterns of Computation

Pattern: foldr (⊕) z [x_1, x_2, …, x_{n-1}, x_n] = x_1 ⊕ (x_2 ⊕ … (x_{n-1} ⊕ (x_n ⊕ z)) …)

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (⊕) z (x: xs) = x ⊕ foldr (⊕) z xs          --foldr:
foldr (⊕) z [ ] = z                                --foldr[ ]
```

Pattern: map f [x_1, x_2, … x_n] = [f x_1, f x_2, … f x_n]

```
map :: (a -> b) -> [a] -> [b]
map f (x : xs) = (f x) : map f xs                  --map:
map f [ ] = [ ]                                    --map[ ]
```

Pattern: zipWith b [x_1, x_2, … x_n] [y_1, y_2, … y_n] = [b x_1 y_1, b x_2 y_2, … b x_n y_n]

Note: extra elements in either sequence are dropped

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith b (x:xs) (y:ys) = (b x y): (zipWith b xs ys)     --zipW:
zipWith b [ ]  ys = [ ]                                  --zipW[ ]L
zipWith b xs  [ ] = [ ]                                  --zipW[ ]R
```

Pattern: iterate f x = [x, f x, f(f x), f(f(f x)), …]

```
iterate :: (a -> a) -> a -> [a]
iterate f x =  x : (iterate f (f x))               --iterate
```

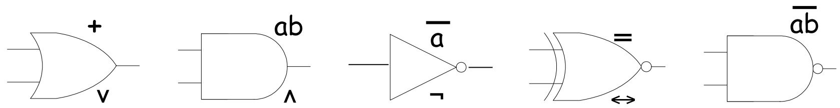
5

Truth Tables for Logical Operators

P	Q	P ∧ Q	P ∨ Q	P ⊗ Q	P→Q	P↔Q	¬ P
False	False	False	False	False	True	True	True
False	True	False	True	True	True	False	
True	False	False	True	True	False	False	
True	True	True	True	False	True	True	False

Combinational Gate Symbols for Logical Operators

with conventional and EE notation for operations



6

Inference Rules: Propositional Calculus

$$\frac{a \quad b}{a \wedge b}\{\wedge I\} \qquad \frac{a \wedge b}{a}\{\wedge E_L\} \qquad \frac{a \wedge b}{b}\{\wedge E_R\}$$

$$\frac{a}{a \vee b}\{\vee I_L\} \qquad \frac{b}{a \vee b}\{\vee I_R\} \qquad \frac{a \vee b \quad a \vdash c \quad b \vdash c}{c}\{\vee E\}$$

$$\frac{a \vdash b}{a \to b}\{\to I\} \qquad \frac{a \quad a \to b}{b}\{\to E\}$$

$$\frac{a}{a}\{ID\} \qquad \frac{\text{False}}{a}\{CTR\} \qquad \frac{\neg a \vdash \text{False}}{a}\{RAA\}$$

7

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
- ∀x.P(x) – ∀ quantifier converts predicate to proposition
  - False if and only if there is some x for which P(x) is False
- ∃x.P(x) – ∀ 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
    - ✓ ∀x. *e* — x is bound in the formula ∀x. *e*
    - ✓ ∃x. *e* — x is bound in the formula ∃x. *e*
  - Free variables are variables that are not bound
- Arbitrary variables in proofs
  - A free variable in a predicate calculus formula is **arbitrary** in a proof if it does not occur free in any undischarged assumption of that proof

8

Inference Rules: Predicate Calculus and Induction

Renaming Variables

$$\frac{F(x)\ \{x \text{ arbitrary}\}}{F(y)}\{R\}$$

$$\frac{\forall x.\ F(x) \quad \{y \text{ not in } F(x)\}}{\forall y.\ F(y)}\{\forall R\}$$

$$\frac{\exists x.\ F(x) \quad \{y \text{ not in } F(x)\}}{\exists y.\ F(y)}\{\exists R\}$$

Introducing/Eliminating Quantifiers

$$\frac{F(x)\ \{x \text{ arbitrary}\}}{\forall x.\ F(x)}\{\forall I\}$$

$$\frac{\forall x.\ F(x)\ \ \{\text{universe is not empty}\}}{F(x)}\{\forall E\}$$

$$\frac{F(x)}{\exists x.\ F(x)}\{\exists I\}$$

$$\frac{\exists x.\ F(x) \quad F(x) \vdash A\ \{x \text{ not free in } A\}}{A}\{\exists E\}$$

Induction

$$\frac{P(0) \quad \forall n.(P(n) \to P(n+1))}{\forall n.P(n)}\{Ind\}$$

Strong Induction

$$\frac{\forall n.((\forall m{<}n.P(m)) \to P(n))}{\forall n.P(n)}\{StrInd\}$$

9

Some Theorems in Rule Form

$$\frac{a \wedge b}{b \wedge a}\{\wedge Comm\}$$

And Commutes

$$\frac{a \vee b}{b \vee a}\{\vee Comm\}$$

Or Commutes

$$\frac{}{a \vee (\neg a)}\{noMiddle\}$$

Law of Excluded Middle

$$\frac{a \to b \quad b \to c}{a \to c}\{\to Chain\}$$

Implication Chain Rule

$$\frac{\neg(a \vee b)}{\neg(b \vee a)}\{\neg(\vee)Comm\}$$

Not Or Commutes

$$\frac{a \to b \qquad \neg b}{\neg a}\{modTol\}$$

Modus Tollens

$$\frac{a \to b}{(\neg b) \to (\neg a)}\{conPos_F\}$$

Contrapositive Fwd

$$\frac{a \qquad \neg a}{False}\{\&\text{-}\}$$

NeverBoth

$$\frac{a \to b}{(\neg a) \vee b}\{\to_F\}$$

Implication Fwd

$$\frac{(\neg a) \vee b}{a \to b}\{\to_B\}$$

Implication Bkw

10

More Theorems in Rule Form

$$\frac{\neg(a \vee b)}{(\neg a)\wedge(\neg b)}\{DeM\vee_F\}$$

DeMorgan Or Fwd

$$\frac{(\neg a)\wedge(\neg b)}{\neg(a \vee b)}\{DeM\vee_B\}$$

DeMorgan Or Bkw

$$\frac{\neg(a \wedge b)}{(\neg a)\vee(\neg b)}\{DeM\wedge_F\}$$

DeMorgan And Fwd

$$\frac{(\neg a)\vee(\neg b)}{\neg(a \wedge b)}\{DeM\wedge_B\}$$

DeMorgan And Bkw

$$\frac{a \vee b \quad \neg a}{b}\{disjSyll\}$$

Disjunctive Syllogism

$$\frac{\neg(\neg a)}{a}\{\neg\ \neg_F\}$$

Double Negation Fwd

$$\frac{a}{\neg(\neg a)}\{\neg\ \neg_B\}$$

Double Negation Bkw

11

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), …

$$\frac{P(n)\ \{n\ arbitrary\}}{\forall n.\ P(n)}\{\forall I\}$$

∀ Introduction

$$\frac{P(0) \quad \forall n.P(n)\rightarrow P(n+1)}{\forall n.\ P(n)}\{Ind\}$$

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
    - ✓ The proof of P(n+1) can cite P(n), P(n-1), … and/or P(0)

12

Haskell Type Specifications

- `x, y, z :: Integer` -- x, y, and z have type Integer
- `xs, ys :: [Integer]` -- sequences with Integer elements
- `xy :: (Integer, Bool)` -- 2-tuple with 1st component Integer, 2nd Bool
- `or :: [Bool] -> Bool` -- function with one argument
  argument is sequence with Bool elems
  delivers value of type Bool
- `(++) :: [e] -> [e] -> [e]` -- 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
- `sum :: Num n => [n] -> n` -- 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

13

Sets

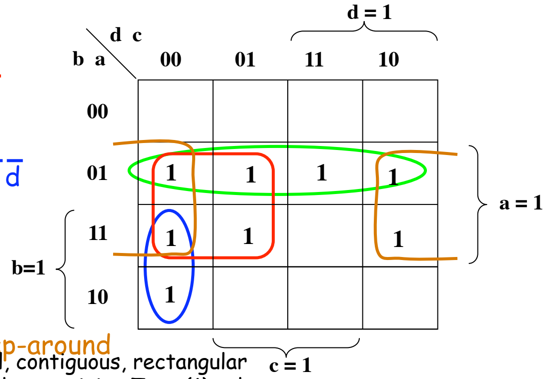
- $\{2, 3, 5, 7, 11\}$ — explicit enumeration
- $2 \in \{2, 3, 5, 7, 11\}$ — stylized epsilon means "element of"
- $\varnothing = \{\ \}$ — stylized Greek letter phi denotes empty set
- $\{x \mid p\ x\}$ — set comprehension
  - Denotes set with elements x, where (p x) is True
- $\{f\ x \mid p\ x\}$ — set comprehension
  - Denotes set with elements of form (f x), where (p x) is True
- $A \subseteq B \leftrightarrow \forall x. (x \in A \rightarrow x \in B)$ — subset
- $A = B \leftrightarrow (A \subseteq B) \wedge (B \subseteq A)$ — set equality
- $A \cup B = \{x \mid x \in A \vee x \in B\}$ — union
- $\bigcup S = \{x \mid \exists A \in S.\ x \in A\}$ — big union
- $A \cap B = \{x \mid x \in A \wedge x \in B\}$ — intersection
- $\bigcap S = \{x \mid \forall A \in S.\ x \in A\}$ — big intersection
- $A - B = \{x \mid x \in A \wedge x \notin B\}$ — set difference
- $A' = U - A$ — complement (U = universe)
- $\mathcal{P}(A) = \{S \mid S \subseteq A\}$ — power set
- $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$ — Cartesian product

14

Karnaugh-Map Minimization Method

$F(a,b,c,d) = a\,\bar{b}\,\bar{c}\,\bar{d} + a\,\bar{b}\,c\,\bar{d} + a\,\bar{b}\,c\,d + a\,b\,\bar{c}\,\bar{d} + a\,b\,c\,\bar{d} + a\,b\,\bar{c}\,d + \bar{a}\,b\,\bar{c}\,\bar{d} + a\,\bar{b}\,\bar{c}\,d$

$= a\,\bar{d} + a\,\bar{b} + b\,\bar{c}\,\bar{d} + a\,\bar{c}$



1. Group together the maximal, contiguous, rectangular regions with 2^k adjacent cells containing True (1) values
2. There is one minterm per cell, so 2^k minterms in all for group
3. Gray-code ordering arranges it so that (n – k) variables have identical form throughout the group ($\bar{x}_j$ in all terms of group or x_j in all of them)
4. Use the distributive law to factor the 2^k minterms into this form: $w = (v_1 + \bar{v}_1)(v_2 + \bar{v}_2) \ldots (v_k + \bar{v}_k)\, v_{k+1}\, v_{k+2} \ldots v_n$
5. Possible because Gray-code ordering puts the other k variables through all possible combinations

15

Running Sum as Sequential Circuit

tail recursion —no stack—

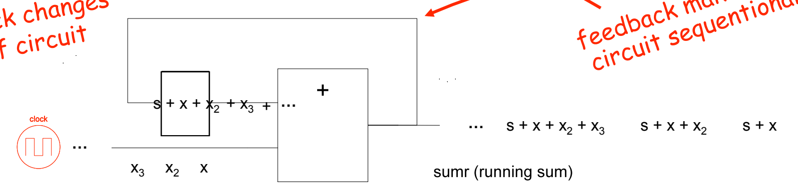
stack-free recursion → "simple" sequential circuit

sumr s (x: xs) = sumr (s + x) xs (sumr).:

sumr s [x] = s (sumr).[]

feedback changes state of circuit

feedback makes this circuit sequential



- ~~Theorem: sumr s [x_1, x_2, … x_n] = s + sum[x_1, x_2, … x_n]~~
  - ✓ But … no output from circuit until at an input arrives
  - ✓ So, theorem applies only when list is nonempty
- What is the proper statement of theorem for circuit?
  - ✓ Circuit theorem: sumr s [x_1, x_2, … x_{n+1}] = s + sum[x_1, x_2, … x_{n+1}]

16

Computation Time and Big O Notation

- Axioms for dmx

```
dmx [ ] = ( [ ] , [ ] )                          --dmx[ ]
dmx [x] = ( [x] , [ ] )                          --dmx[x]
dmx (x1 : (y1 : xys)) = ( x1 : xs , y1 : ys )    --dmx::
  where (xs, ys) = dmx xys
```

- Computation time for dmx
  - T_n = time required for to compute dmx$[x_1, x_2, \ldots x_n]$
- Recurrence equations
  - $T_0 = T_1 = 3$ 3 ops: matching, [ ]-build, pair-build
  - $T_n = T_{n-2} + 4$ 4 ops: matching, 2 insertions, pair-build plus deal sequence that is shorter by 2
  - $T_n \le 4n, \forall n > 0$ that is, $T_n = O(n)$ — prove by induction
- Bounding the rate of growth
  - *Given: functions f and g*
  - *f is big-O of g, written f = O(g), means*
    - ✓ $\exists c, s.\ \forall x > s.\ f(x) \le c \cdot g(x)$

17

Search Trees

- Search trees – formal representation
  - data SearchTree key dat = Nub | Cel key dat (SearchTree key dat) (SearchTree key dat)
- Axioms for height of tree

```
height Nub = 0                                     --height Nub
height (Cel k d left right) =
           1 + max (height left) (height right)    --height Cel
```

  - Serves as both inductive definition and computer program – Why?
    - ✓ Correct
    - ✓ Cover all cases
    - ✓ Inductive parts are closer to non-inductive case
- Theorem (logarithmic height)
  - *A SearchTree of height h can contain $2^h - 1$ items*
  - n items can be stored in a SearchTree of height $\lceil \log_2(n+1) \rceil$
  - Proof — induction on height
  - Conclusion — in a well constructed SearchTree, retrieval time is proportional to the logarithm of the number of items

18

Order/Balance Axioms & Key Insertion Property

- Axioms for balanced-tree predicate (inductive definition)

```
balanced Nub = True                                          --balanced Nub
balanced (Cel k d left right) = (abs((height left) – (height right))<=1)
            /\ (balanced left)  /\  (balanced right)        --balanced Cel
```

- Axioms for ordered-tree predicate (inductive definition)

```
ordered Nub = True                                           --ordered Nub
ordered (Cel k d (Cel x c xL xR) (Cel y b yL yR)) = (k > x) /\ (k < y) /\
   (ordered(Cel x c xL xR)) /\ (ordered Cel y b yL yR))     --ordered Cels
ordered (Cel k d (Cel x c xL xR) Nub) = (k > x) /\
   (ordered(Cel x c xL xR))                                  --ordered CCN
ordered (Cel k d Nub (Cel y b yL yR)) = (k < y) /\
   (ordered(Cel y b yL yR))                                  --ordered CNC
ordered (Cel k d Nub Nub) = True                             --ordered CNN
```

- Insertion operator (^:) must preserve order and balance and conserve keys

```
∀s.∀k.∀d. ordered s  →  ordered ((k, d) ^: s)
∀s.∀k.∀d. balanced s  →  balanced ((k, d) ^: s)
(((y = x) ∨ (y∈s)) ∧ ordered(s) ∧ balanced(s)) ↔
   ((y∈((x, a)^:s)) ∧ ordered((x, a)^:s) ∧ balanced((x, a)^:s))
```

Inductive Definition of Tree-Insertion

- (x, a) ^: s

```
(x, a) ^: Nub = (Cel x a Nub Nub)                          --^:Nub
(x, a) ^: (Cel z c left right) =                           --^:Cel
   if x < z
    then if (height newLeft ) > (height right) + 1
         then rotR(Cel z c  newLeft  right)
         else (Cel z c newLeft right)
   else if x > z
    then if (height newRight ) > (height left) + 1
         then rotL(Cel z c left newRight)
         else (Cel z c left newRight)
   else (Cel z a left right)
   where
      newLeft  = (x, a) ^: left
      newRight = (x, a) ^: right
```

20

Retrieving Data from a Search Tree

- Found or Not Found

  Example, item found: Just (2088, "LaserJet")

  data Maybe a = Just a | Nothing ← Not-Found Indicator
- Definition "occurs in"
  - s :: SearchTree key dat, k :: key, d :: dat
  - k occurs in s — that is, $k \in s$

    $k \in s \leftrightarrow (\exists x,d,left,right.\ ((s = (Cel\ x\ d\ left\ right)) \wedge (k = x \vee k \in left \vee k \in right)))$
- Axioms for getItem

```
getItem (Cel key dat smaller bigger) searchKey =
   if  searchKey < key then (getItem smaller searchKey)      --g<
   else if searchKey > key then (getItem bigger  searchKey)  --g>
   else (Just(key, dat))                                     --g=
getItem Nub searchKey = Nothing                              --gNub
```

- Theorem (getItem)

  $((\text{ordered } s) \wedge k \in s) \rightarrow \text{getItem } s\ k = \text{Just } (k, d)$

21

Tree Induction — another proof method

Tree Induction:

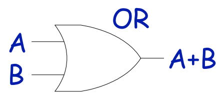
$$\frac{\forall s.((\forall r \subset s.P(r)) \rightarrow P(s))}{\forall s.P(s)}\ \{TreeInd\}$$

- Definitions
  - Subtree
    - ✓ s, t :: SearchTree key dat
    - ✓ $s \subseteq t \leftrightarrow (s = t) \vee (\exists k,d,left,right.\ (t = Cel\ k\ d\ left\ right)) \wedge ((s \subseteq left) \vee (s \subseteq right))$
  - Proper Subtree
    - ✓ s, t :: SearchTree key dat
    - ✓ $s \subset t \leftrightarrow (\exists k,d,left,right.\ (t = Cel\ k\ d\ left\ right)) \wedge ((s \subseteq left) \vee (s \subseteq right))$
    - ✓ Equivalent Definition: $s \subset t \leftrightarrow s \subseteq t \wedge s \neq t$
- Tree induction
  - P — predicate parameterized over SearchTrees
    - ✓ P(t) is a proposition whenever t :: SearchTree key dat
  - Prove:
    - ✓ Base case: P(Nub)
    - ✓ Inductive case: P(Cel z c lf rt) —assume P(s) if $s \subset$ Cel z c lf rt
  - Conclude: $\forall t.\ P(t)$

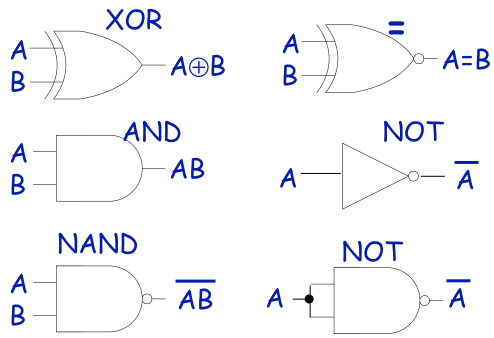
22

Logical Operators

P	Q	P ∧ Q	P ∨ Q	P ⊻ Q	P→Q	P↔Q	¬ P
False	False	False	False	False	True	True	True
False	True	False	True	True	True	False	
True	False	False	True	True	False	False	
True	True	True	True	False	True	True	False



INPUTS		OUTPUT
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1



23

Binary Arithmetic

Axioms

natFromDgts b [ ] = 0 {nat[ ]}
natFromDgts b (d: ds) = d + b*(natFromDgts b ds) {nat:}
dgtsFromNat b 0 = [ ] {dgts0}
dgtsFromNat b n = (n `mod` b):(dgtsFromNat b (n `div` b)) {dgts>0}
num = natFromDgts 2 {num}
bits = dgtsFromNat 2 {bits}
twos w n = if n >= 0 then (bits n) else (bits(n + 2^w)) {2s}
word w n = take w ((twos w n) ++ (repeat 0)) {word}
adder [ ] [ ] c_0 = ([ ], c_0) {adder[ ]}
adder (x_0: xs) (y_0: ys) c_0 = (s_0: ss, c) {adder::}
  where
  [s_0, c_1] = fullAdder x_0 y_0 c_0
  (ss, c) = adder xs ys c_1

Derived properties

$I(w) = \{n \mid -2^{w-1} \le n < 2^{w-1}\}$

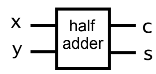
num[ ] = 0 {num[ ]}
num(b: bs) = b + 2*(num bs) {num :}
bits 0 = [ ] {bits0}
bits(n+1) = ((n+1) `mod` 2): (bits((n+1) `div` 2)) {bits>0}
∀w∈N. ∀n∈I(w). (length(twos w n) ≤ w) {2s fits}
∀w.∀n∈N. num(word w n) = n mod 2^w {word=N mod 2^w}
∀w.∀n∈I(w). num(word w n) = n mod 2^w {word=I(w) mod 2^w}
∀w.∀n∈I(w).(word w (-n)) = word w (1 + num(map (1-) (word n)) {2s trick}
∀w.∀x,y∈I(w). adder (word w x) (word w y) = word w (x + y) {adder}

24

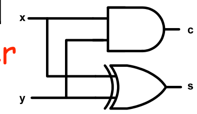
One-Bit Adder Circuits

half-adder

x+y	cs
0+0	00
0+1	01
1+0	01
1+1	10



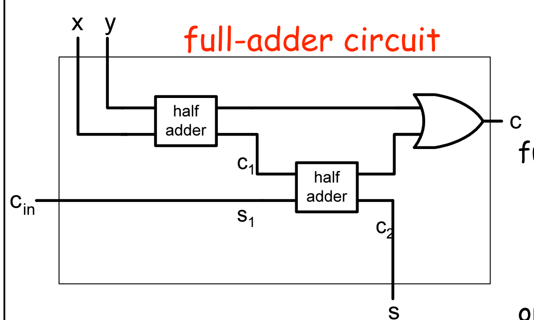
half-adder circuit

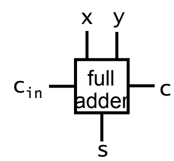


half-adder model

```
signal bool = if bool then 1 else 0
and01 x y = signal((x==1) /\ (y==1))
xor01 x y = signal(x/=y)
halfAdder x y = [xor01 x y, and01 x y]
```

full-adder circuit





full-adder

c_{in}+x+y	cs
0+0+0	00
0+0+1	01
0+1+0	01
0+1+1	10
1+0+0	01
1+0+1	10
1+1+0	10
1+1+1	11

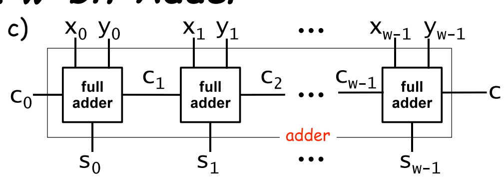
full-adder model

fullAdder x y c_{in} = [s, c]
  where
  [s_1, c_1] = halfAdder x y
  [s, c_2] = halfAdder s_1 c_{in}
  c = or01 c_1 c_2
or01 x y = signal((x==1) \/ (y==1))
signal bool = if bool then 1 else 0

25

Twos Complement Arithmetic
with w-bit Adder

adder (x_0: xs) (y_0: ys) c_0 = (s_0: ss, c)
  where
  [s_0, c_1] = fullAdder x_0 y_0 c_0
  (ss, c) = adder xs ys c_1
adder [ ] [ ] c_0 = ([ ], c_0)



Theorem (w-bit adder)

$\forall w.\ \forall c_0 \in \{0,1\}.\ \forall [x_0, x_1, \ldots x_{w-1}], [y_0, y_1, \ldots y_{w-1}] \in [\{0,1\}].$
$((([s_0, s_1, \ldots s_{w-1}], c) = (\text{adder } [x_0, x_1, \ldots x_{w-1}]\ [y_0, y_1, \ldots y_{w-1}]\ c_0))$
$\rightarrow (\text{nat}([s_0, s_1, \ldots s_{w-1}]{+}{+}[c]) = (\text{nat}[x_0, x_1, \ldots x_{w-1}] + \text{nat}[y_0, y_1, \ldots y_{w-1}] + c_0)))$

- 2s complement arithmetic
  - Just throw away c (output carry)
  - Numerically, this reduces value by $c*2^w$
    - ✓ That gives (nat[s_0, s_1, … s_{w-1}] mod 2^w)
    - ✓ Exactly what is needed for 2s complement arithmetic
      - Consequence of (word=$\mathcal{I}$(w) mod 2^w) theorem
- Theorem (w-bit adder works):
  $\forall w. \forall x,y \in \mathcal{I}(w). \exists c.(\text{adder (word w x) (word w y) 0}) = (\text{word w (x + y), c})$

26