1.3 Case study: boolean formulas

type Name = String

data Form = F | T
type Name = String

data Form = F | T
  | Var Name
  | Not Form
  | And Form Form
  | Or Form Form
### 1.3 Case study: boolean formulas

```haskell
import Data.Tuple (swap)

-- Internal representation
data Form = F | T | Var Name | Not Form | And Form Form | Or Form Form deriving Eq

-- Pretty printing
par :: String -> String
par s = "(" ++ s ++ ")"
```

Example: `Or (Var "p") (Not(Var "p"))`

More readable: symbolic infix constructors, must start with `:`
```haskell
data Form = F | T | Var Name | Not Form | Form &: Form | Form :|: Form deriving Eq
```

Now: `Var "p" :|: Not(Var "p")`
par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Form where
    show F = "F"
    show T = "T"
    show (Var x) = x
    show (Not p) = par("~" ++ show p)
    show (p :&: q) = par(show p ++ " & " ++ show q)
    show (p :+: q) = par(show p ++ " | " ++ show q)
par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Form where
    show F = "F"
    show T = "T"
    show (Var x) = x
    show (Not p) = par("~" ++ show p)

Form is the syntax of boolean formulas, not their meaning:

\[ \text{Not(Not } T \text{) and } T \text{ mean the same} \]

Form is the syntax of boolean formulas, not their meaning:

\[ \text{Not(Not } T \text{) and } T \text{ mean the same but are different:} \]

\[ \text{Not(Not } T \text{) } \not= T \]
Syntax versus meaning

Form is the syntax of boolean formulas, not their meaning:

\[ \text{Not(Not T)} \text{ and } T \text{ mean the same but are different:} \]

\[ \text{Not(Not T) /= T} \]

What is the meaning of a Form?

Its value!? 

But what is the value of \( \text{Var "p"} \) ?

```
-- Wertebelegung
type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
```

```
-- Wertebelegung
type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
```
type Valuation = [(Name, Bool)]

eval :: Valuation -> Form -> Bool

eval _ F = False

eval _ T = True

eval v (Var x) = the(lookup x v) where the(Just b) = b

eval v (Not p) = not(eval v p)
-- Wertebelegung

type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
eval v (Var x) = the(lookup x v) where the(Just b) = b
eval v (Not p) = not(eval v p)
val v (p :&: q) = eval v p && eval v q
val v (p :+: q) = eval v p || eval v q

-- Wertebelegung

type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
eval v (Var x) = the(lookup x v) where the(Just b) = b
eval v (Not p) = not(eval v p)
val v (p :&: q) = eval v p && eval v q
val v (p :+: q) = eval v p || eval v q

-- Wertebelegung

type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
eval v (Var x) = the(lookup x v) where the(Just b) = b
eval v (Not p) = not(eval v p)
val v (p :&: q) = eval v p && eval v q
val v (p :+: q) = eval v p || eval v q

> eval ["a",False], ("b",False)]
(Not(Var "a") :&: Not(Var "b"))
All valuations for a given list of variable names:

vals :: [Name] -> [Valuation]
vals [] = [[]]
vals (x:xs) = [ (x, False):v | v <- vals xs ] ++
            [ (x, True):v | v <- vals xs ]

vals ["b"]
= [ ("b", False):v | v <- vals [] ] ++
  [ ("b", True):v | v <- vals [] ]
All valuations for a given list of variable names:

```haskell
vals :: [Name] -> [Valuation]
vals [] = [[]]
vals (x:xs) = [(x,False):v | v <- vals xs] ++
   [(x,True):v | v <- vals xs]

vals [
  "b"
] = [("b",False):v | v <- vals []] ++
   [("b",True):v | v <- vals []]
= [("b",False):[] ++ [("b",True):[]
= [("b",False), ("b",True)]

vals [
  "a","b"
] = [("a",False):v | v <- vals ["b"]] ++
   [("a",True):v | v <- vals ["b"]]
= [("a",False), ("b",False)] ++ [("a",False), ("b",True)] +
   [("a",True), ("b",False)] ++ [("a",True), ("b",True)]
```
All valuations for a given list of variable names:

vals :: [Name] -> [Valuation]
vals [] = [[]]
vals (x:xs) = [(x,False):v | v <- vals xs] ++
               [(x,True):v | v <- vals xs]

vals ["b"]
  = [(["b",False]:v | v <- vals []]] ++
    [(["b",True]:v | v <- vals [])]
  = [(["b",False]:[])] ++ [(["b",True]:[]]
  = [(["b",False), (["b",True])]

vals ["a","b"]
  = [(["a",False]:v | v <- vals ["b"]]] ++
    [(["a",True]:v | v <- vals ["b"]]
  = [(["a",False), (["b",False]) ++ [(["a",False), (["b",True]) +
    [(["a",True), (["b",False]) ++ [(["a",True), (["b",True])

Does vals construct all valuations?

prop_vals1 xs =
  length(vals xs) ==

Does vals construct all valuations?

prop_vals1 xs =
  length(vals xs) == 2 ^ length xs
Does vals construct all valuations?

```
prop_vals1 xs = 
    length(vals xs) == 2 ^ length xs

prop_vals2 xs = 
distinct (vals xs)

distinct :: Eq a => [a] -> Bool
distinct [] = True
distinct (x:xs) = not(elem x xs) && distinct xs
```

Demo

Restrict size of test cases:

```
prop_vals1' xs = 
    length xs <= 10 ==> 
    length(vals xs) == 2 ^ length xs

prop_vals2' xs = 
    length xs <= 10 ==> distinct (vals xs)
```
Restrict size of test cases:

```haskell
prop_vals1' xs =
  length xs <= 10 ==> .
length(vals xs) == 2 ^ length xs
```

```haskell
prop_vals2' xs =
  length xs <= 10 ==> distinct (vals xs)
```

**Demo**
Restrict size of test cases:

\[
\text{prop_vals1' } xs = \\
\quad \text{length } xs \leq 10 \implies \\
\quad \text{length(vals } xs) = 2 \cdot \text{length } xs
\]

\[
\text{prop_vals2' } xs = \\
\quad \text{length } xs \leq 10 \implies \text{distinct (vals } xs)
\]

Demo

---

Satisfiable and tautology

\[
satisfiable :: \text{Form} \to \text{Bool}
\]

\[
satisfiable p = \text{or } [\text{eval } v \ p \mid v \leftarrow \text{vals} (\text{vars } p)]
\]

Satisfiable and tautology

\[
satisfiable :: \text{Form} \to \text{Bool}
\]

\[
satisfiable p = \text{or } [\text{eval } v \ p \mid v \leftarrow \text{vals} (\text{vars } p)]
\]

\[
tautology :: \text{Form} \to \text{Bool}
\]

\[
tautology = \text{not } . \, \text{satisfiable} . \, \text{not}
\]
Satisfiable and tautology

satisfiable :: Form -> Bool
satisfiable p = or [eval v p | v <- vals vars p]

tautology :: Form -> Bool
tautology = not . satisfiable . Not

vars :: Form -> [Name]

Satisfiable and tautology

satisfiable :: Form -> Bool
satisfiable p = or [eval v p | v <- vals vars p]

tautology :: Form -> Bool
tautology = not . satisfiable . Not

vars :: Form -> [Name]
vars F = []
vars T = []
vars (Var x) = [x]
vars (Not p) = vars p
vars (p :&: q) = nub (vars p ++ vars q)
vars (p :+: q) = nub (vars p ++ vars q)

Simplifying a formula: Not inside?

p0 :: Form
p0 = (Var "a" :&: Var "b") :+: (Not (Var "a") :&: Not (Var "b"))
Simplifying a formula: Not inside?

\[ \text{isSimple} :: \text{Form} \rightarrow \text{Bool} \]
\[ \text{isSimple} \ (\text{Not} \ p) \ = \ \text{not} \ (\text{isOp} \ p) \]

where
\[ \text{isOp} \ (\text{Not} \ p) \ = \ \text{True} \]
\[ \text{isOp} \ (p :\&: q) \ = \ \text{True} \]
\[ \text{isOp} \ (p :\|: q) \ = \ \text{True} \]

Simplifying a formula: Not inside?

\[ \text{isSimple} :: \text{Form} \rightarrow \text{Bool} \]
\[ \text{isSimple} \ (\text{Not} \ p) \ = \ \text{not} \ (\text{isOp} \ p) \]

where
\[ \text{isOp} \ (\text{Not} \ p) \ = \ \text{True} \]
\[ \text{isOp} \ (p :\&: q) \ = \ \text{True} \]
\[ \text{isOp} \ (p :\|: q) \ = \ \text{True} \]
\[ \text{isOp} \ p \ = \ \text{False} \]
Simplifying a formula: Not inside?

isSimple :: Form -> Bool
isSimple (Not p) = not (isOp p)
  where
  isOp (Not p) = True
  isOp (p &: q) = True
  isOp (p :+: q) = True
  isOp p = False
isSimple (p &: q) = isSimple p && isSimple q
isSimple (p :+: q) = isSimple p && isSimple q
isSimple p = True

Simplifying a formula: Not inside!

simplify :: Form -> Form
simplify (Not p) = pushNot (simplify p)
  where
  pushNot (Not p) = p
simplify :: Form -> Form
simplify (Not p)  =  pushNot (simplify p)
where
  pushNot (Not p)  =  p
  pushNot (p :&: q)  =  pushNot p :&: pushNot q
  pushNot (p :|| q)  =  pushNot p :|| pushNot q
  pushNot p  =  Not p
Simplifying a formula: Not inside!

\[
simplify : \text{Form} \to \text{Form} \\
simplify \left( \text{Not } p \right) = \text{pushNot} \left( \text{simplify } p \right) \\
\text{where} \\
\text{pushNot} \left( \text{Not } p \right) = p \\
\text{pushNot} \left( p \land q \right) = \text{pushNot } p \lor \text{pushNot } q \\
\text{pushNot} \left( p \lor q \right) = \text{pushNot } p \land \text{pushNot } q \\
\text{pushNot } p = \text{Not } p \\
simplify \left( p \land q \right) = \text{simplify } q \land \text{simplify } q \\
simplify \left( p \lor q \right) = \text{simplify } p \lor \text{simplify } q
\]

QuickCheck

-- for QuickCheck: test data generator for Form

instance Arbitrary Form where

arbitrary = sized prop

where

prop 0 = oneof [return F, return T, liftM Var arbitrary]

prop n | n > 0 = oneof

[return F, return T, liftM Var arbitrary, liftM Not (prop (n-1)), liftM2 (\&\&) (prop n 'div' 2) (prop n 'div' 2), liftM2 (\|\|) (prop n 'div' 2) (prop n 'div' 2)]

prop_simplify p = isSimple (simplify p)
Simplifying a formula: Not inside!

\[
\begin{align*}
simplify & :: \text{Form} \to \text{Form} \\
simplify \ (\text{Not} \ p) & = \ \text{pushNot} \ (\simplify \ p) \\
\text{where} & \\
\text{pushNot} \ (\text{Not} \ p) & = \ p \\
\text{pushNot} \ (p :\&: q) & = \ \text{pushNot} \ p :\|: \text{pushNot} \ q \\
\text{pushNot} \ (p :\|: q) & = \ \text{pushNot} \ p :\&: \text{pushNot} \ q \\
\text{pushNot} \ p & = \ \text{Not} \ p \\
\simplify \ (p :\&: q) & = \ \simplify \ q :\&: \ \simplify \ q \\
\simplify \ (p :\|: q) & = \ \simplify \ q :\|: \ \simplify \ q \\
\simplify \ p & = \ p
\end{align*}
\]

8.4 Structural induction

Structural induction for Tree

\[
data \ \text{Tree} \ a = \ \text{Empty} \mid \ \text{Node} \ a \ (\text{Tree} \ a) \ (\text{Tree} \ a)
\]

Structural induction for Tree

\[
data \ \text{Tree} \ a = \ \text{Empty} \mid \ \text{Node} \ a \ (\text{Tree} \ a) \ (\text{Tree} \ a)
\]

To prove property \( P(t) \) for all finite \( t :: \text{Tree} \ a \).
Structural induction for Tree

data Tree a = Empty | Node a (Tree a) (Tree a)

To prove property $P(t)$ for all finite $t :: Tree a$
Base case: Prove $P(Empty)$ and
Induction step: Prove $P(Node x t1 t2)$ assuming the induction hypotheses $P(t1)$ and $P(t2)$.

Example

flat :: Tree a -> [a]
flat Empty = []
flat (Node x t1 t2) =
    flat t1 ++ [x] ++ flat t2

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Empty = Empty
mapTree f (Node x t1 t2) =
    Node (f x) (mapTree f t1) (mapTree f t2)

Lemma flat (mapTree f t) = map f (flat t)
Lemma $\text{flat} \ (\text{mapTree} \ f \ t) = \text{map} \ f \ (\text{flat} \ t)$

Proof by structural induction on $t$

Induction step:

$IH1$: $\text{flat} \ (\text{mapTree} \ f \ t1) = \text{map} \ f \ (\text{flat} \ t1)$

$IH2$: $\text{flat} \ (\text{mapTree} \ f \ t2) = \text{map} \ f \ (\text{flat} \ t2)$

To show: $\text{flat} \ (\text{mapTree} \ f \ (\text{Node} \ x \ t1 \ t2)) = \text{map} \ f \ (\text{flat} \ (\text{Node} \ x \ t1 \ t2))$

$\text{flat} \ (\text{mapTree} \ f \ (\text{Node} \ x \ t1 \ t2))$

$= \text{flat} \ (\text{Node} \ (f \ x) \ (\text{mapTree} \ f \ t1) \ (\text{mapTree} \ f \ t2))$
Lemma \( \text{flat} (\text{mapTree } f \ t) = \text{map } f (\text{flat } t) \)

Proof by structural induction on \( t \)

Induction step:

IH1: \( \text{flat} (\text{mapTree } f \ t_1) = \text{map } f (\text{flat } t_1) \)

IH2: \( \text{flat} (\text{mapTree } f \ t_2) = \text{map } f (\text{flat } t_2) \)

To show: \( \text{flat} (\text{mapTree } f (\text{Node } x \ t_1 \ t_2)) = \text{map } f (\text{flat } (\text{Node } x \ t_1 \ t_2)) \)

\[
\begin{align*}
\text{flat} (\text{mapTree } f (\text{Node } x \ t_1 \ t_2)) &= \text{flat} (\text{Node } (f \ x) (\text{mapTree } f \ t_1) (\text{mapTree } f \ t_2)) \\
&= \text{flat} (\text{mapTree } f \ t_1) ++ [f \ x] ++ \text{flat} (\text{mapTree } f \ t_2) \\
&= \text{map } f (\text{flat } t_1) ++ [f \ x] ++ \text{map } f (\text{flat } t_2) \\
&\quad \text{ -- by IH1 and IH2} \\
&\text{map } f (\text{flat } (\text{Node } x \ t_1 \ t_2)) \\
&= \text{map } f (\text{flat } t_1 ++ [x] ++ \text{flat } t_2) \\
&= \text{map } f (\text{flat } t_1) ++ [f \ x] ++ \text{map } f (\text{flat } t_2)
\end{align*}
\]

The general (regular) case

data \( T \ a = \ldots \)

Assumption: \( T \) is a regular data type:
The general (regular) case

data T a = ... 

Assumption: T is a regular data type:
    Each constructor $C_i$ of T must have a type
    $t_1 \rightarrow \ldots \rightarrow t_m \rightarrow T \ a$
    such that each $t_j$ is either $T \ a$ or does not contain T

To prove property $P(t)$ for all finite $t :: T \ a$:
    prove for each constructor $C_i$ that $P(C, x_1 \ldots x_n)$
    assuming the induction hypotheses $P(x_j)$ for all $j \ s.t. \ t_j = T \ a$
Structural induction for Tree

data Tree a = Empty | Node a (Tree a) (Tree a)

To prove property $P(t)$ for all finite $t :: Tree a$

Base case: Prove $P(Empty)$ and

Induction step: Prove $P(Node x t1 t2)$

assuming the induction hypotheses $P(t1)$ and $P(t2)$.

• So far, only batch programs:
  given the full input at the beginning,
  the full output is produced at the end

• Now, interactive programs:
  read input and write output
  while the program is running

The problem

• Haskell programs are pure mathematical functions:

The problem

• Haskell programs are pure mathematical functions:
  Haskell programs have no side effects
An impure solution

Most languages allow functions to perform I/O without reflecting it in their type.

Assume that Haskell were to provide an input function

\texttt{inputInt :: Int}

Now all functions potentially perform side effects.

Now we can no longer reason about Haskell like in mathematics.
An impure solution

Most languages allow functions to perform I/O without reflecting it in their type.

Assume that Haskell were to provide an input function

\[ \text{inputInt :: Int} \]

Now all functions potentially perform side effects.
Now we can no longer reason about Haskell like in mathematics:

\[ \text{inputInt - inputInt = 0} \]

The pure solution

Haskell distinguishes expressions without side effects from expressions with side effects (actions) by their type:

\[ \text{IO a} \]

is the type of (I/O) actions that return a value of type a.

Example

\[ \text{Char: the type of pure expressions that return a Char} \]
The pure solution

Haskell distinguishes expressions without side effects from expressions with side effects (actions) by their type:

\[
\text{IO } a
\]
is the type of (I/O) actions that return a value of type \(a\).

Example

- `Char`: the type of pure expressions that return a `Char`
- `IO Char`: the type of actions that return a `Char`

- Type `()` is the type of empty tuples (no fields).
- The only value of type `()` is `()`, the empty tuple.
Type (Unit) is the type of empty tuples (no fields).
The only value of type (Unit) is (), the empty tuple.
Therefore IO (Unit) is the type of actions
that return the dummy value (Unit)

- **getChar :: IO Char**
  - Reads a Char from standard input,
    echoes it to standard output,
    and returns it as the result
- **putChar :: Char -> IO ()**
  - Writes a Char to standard output,
    and returns no result
- **return :: a -> IO a**
Basic actions

- `getChar :: IO Char`
  Reads a Char from standard input, echoes it to standard output, and returns it as the result
- `putChar :: Char -> IO ()`
  Writes a Char to standard output, and returns no result
- `return :: a -> IO a`
  Performs no action, just returns the given value as a result

Sequencing: `do`

A sequence of actions can be combined into a single action with the keyword `do`

Example

```
get2 :: IO ?
get2 = do x <- getChar
```

Sequencing: `do`

A sequence of actions can be combined into a single action with the keyword `do`

Example

```
get2 :: IO ?
get2 = do x <- getChar -- result is named x
    getChar
```
**Sequencing: do**

A sequence of actions can be combined into a single action with the keyword `do`.

**Example**

```plaintext
get2 :: IO ?
get2 = do x <- getChar    -- result is named x
       y <- getChar    -- result is ignored
       return (x, y)
```

**General format (observe layout!):**

```plaintext
do  a_1
  
  a_n
```

where each `a_i` can be one of

- an action
  - Effect: execute action
- `x <- action`
  - Effect: execute action :: IO `a`, give result the name `x :: a`
- `let x = expr`
General format (observe layout!):

do  \( a_1 \)
  :
  \( a_n \)

where each \( a_i \) can be one of

- an action  
  Effect: execute action
- \( x \leftarrow \text{action} \)  
  Effect: execute \( \text{action} :: \text{IO} a \), give result the name \( x :: a \)
- \( \text{let } x = \text{expr} \)  
  Effect: give \( \text{expr} \) the name \( x \)  
  Lazy: \( \text{expr} \) is only evaluated when \( x \) is needed!

Derived primitives

Write a string to standard output:

\[
\text{putStr} :: \text{String} \to \text{IO} ()
\]
\[
\text{putStr} [] = \text{return} ()
\]
Derived primitives

Write a string to standard output:

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (c:cs) = do putChar c
                 putStr cs
```

Write a line to standard output:

```haskell
putStrLn :: IO ()
putStrLn cs = putStrLn (cs ++ '\n')
```

Read a line from standard input:

```haskell
getLine :: IO String
getLine = do x <- getChar
```
Read a line from standard input:

```haskell
getLine :: IO String
getLine = do x <- getChar
            if x == '\n' then
              return []
            else
              do xs <- getLine
              return (x:xs)
```

Read a line from standard input:

```haskell
getLine :: IO String
getLine = do x <- getChar
            if x == '\n' then
              return []
            else
              do xs <- getLine
              return (x:xs)
```

Derived primitives

Write a string to standard output:

```haskell
putStr :: String -> IO ()
```

Actions are normal Haskell values and can be combined as usual, for example with if-then-else.
Prompt for a string and display its length:

```
strLen :: IO ()
strLen = do putStr "Enter a string: "
           xs <- getLine
```
How to read other types

Input string and convert

Useful class:

```haskell
class Read a where
    read :: String -> a
```

Most predefined types are in class Read.

Example:

```haskell
getInt :: IO Integer
getInt = do xs <- getLine
            return (read xs)
```
Case study

The game of Hangman
in file hangman.hs

Does vals construct all valuations?

prop_vals1 xs =
  length(vals xs) == 2 ^ length xs

prop_vals2 xs =
  distinct (vals xs)

distinct :: Eq a => [a] -> Bool
distinct [] = True
distinct (x:xs) = not (elem x xs) && distinct xs

Demo

----------
| | |
| | |
| | |
| | |

Word: ---e--
Missed: e

----------
| | |
| | |
| | |
| | |

Word: --le--
Missed: s s