1.3 Case study: boolean formulas

```haskell
  type Name = String

  data Form = F | T
             | Var Name
             | Not Form
```
1.3 Case study: boolean formulas

type Name = String

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

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

More readable: symbolic infix constructors, must start with :
data Form = F | T | Var Name
  | Not Form
  | Form :&: Form
  | Form :+: Form
deriving Eq
par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Form where
  show F = "F"
  show T = "T"

par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Form where
  show F = "F"
  show T = "T"
  show (Var x) = x
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)

> Var "p" :&: Not(Var "p")
Pretty printing

```haskell
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)
```

```
> Var "p" :&: Not(Var "p")
(p & (~p))
```

Syntax versus meaning

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

```
Not(Not T) and T mean the same
```

Syntax versus meaning

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

```
Not(Not T) and T mean the same but are different:

Not(Not T) /= T
```

What is the meaning of a Form?
Syntax versus meaning

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

Not(Not T) and T mean the same but are different:

Not(Not T) /= T

What is the meaning of a Form?

Its value!?  

But what is the value of Var "p"?  

-- Wertebellegung

\[ \text{type Valuation} = [(\text{Name}, \text{Bool})] \]

eval :: Valuation -> Form -> Bool
-- Wertebeflegung

type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool

eval _ F = False

eval _ T = True

eval v (Var x) = fromJust(lookup x v)

eval v (Not p) = not(eval v p)
type Valuation = [(Name, Bool)]

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

> eval [("a", False), ("b", False)]
  (Not (Var "a") :&: Not (Var "b"))
val `a` = `b`

val `a` = `b`

val `a` = `b`

val `a` = `b`

val `a` = `b`

val `a` = `b`

val `a` = `b`

val `a` = `b`

val `a` = `b`
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)]
```

Does vals construct all valuations?

```haskell
prop_vals1 xs =
  length(vals xs) ==
```

Does vals construct all valuations?

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

prop_vals2 xs =
  distinct (vals xs)
```
Does vals construct all valuations?

```haskell
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
```

Restrict size of test cases:

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

```
Last login: Fri Dec 5 08:09:36 on ttys004
122:~ nipkow$ cd Teaching/FP/1415/Code/
122:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l Form
[1 of 1] Compiling Form                      ( Form.hs, interpreted )
Ok, modules loaded: Form.
*Form> quickCheck prop_vals1_
```

```
Loading package array-0.4.0.1 ... linking ... done.
Loading package deepseq-1.3.0.1 ... linking ... done.
Loading package old-locale-1.0.0.5 ... linking ... done.
Loading package time-1.4.0.1 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
Loading package containers-0.5.0.0 ... linking ... done.
Loading package pretty-1.1.1.0 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.6 ... linking ... done.
(32 tests)
```
Restrict size of test cases:

\[
\text{prop_vals1'} \; \text{xs} = \\
\quad \text{length} \; \text{xs} \leq 10 \\n\quad \text{length(vals} \; \text{xs}) = 2 \, ^\wedge \, \text{length} \; \text{xs}
\]

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

Demo

Satisfiable and tautology

\[
\text{satisfiable : Form} \rightarrow \text{Bool} \\
\text{satisfiable p} = \text{or}[\text{eval v p} | v <\text{vals(vars p)}]
\]

\[
\text{tautology : Form} \rightarrow \text{Bool} \\
\text{tautology} = \text{not} \cdot \text{satisfiable} \cdot \text{Not}
\]

\[
\text{vars : Form} \rightarrow \text{[Name]} \\
\text{vars} \; \text{F} = [] \\
\text{vars} \; \text{T} = [] \\
\text{vars(Var x)} = [x] \\
\text{vars(Not p)} = \text{vars} \; \text{p}
\]
p0 :: Form
p0 = (Var "a" &: Var "b") : (Not (Var "a") &: Not (Var "b"))

> vals (vars p0)
[[("a",False),("b",False)], [("a",False),("b",True)], [("a",True), ("b",False)], [("a",True), ("b",True )]]

> eval v p0 | v <- vals (vars p0) ]
p0 :: Form
p0 = (Var "a" &: Var "b") : |:
    (Not (Var "a") &: Not (Var "b"))

> vals (vars p0)
[["a",False], ["b",False]], [["a",False], ["b",True]],
[["a",True], ["b",False]], [["a",True], ["b",True]]

> [ eval v p0 | v <- vals (vars p0) ]
[True, False, False, True]

> satisfiable p0
True

Simplifying a formula: Not inside?

isSimple :: Form -> Bool

isSimple (Not p) = not (isOp p)
Simplifying a formula: Not inside?

```haskell
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
```
simplify :: Form -> Form
simplify (Not p) = pushNot (simplify 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
-- 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)
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

```
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 $P(t2)$.

($x$, $t1$ and $t2$ are new variables)

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

\textbf{Proof} by structural induction on \( t \)

\text{Induction step:}

1. \( \text{IH1: flat (mapTree \ f \ t1) = map \ f \ (\text{flat} \ t1)} \)
2. \( \text{IH2: flat (mapTree \ f \ t2) = map \ f \ (\text{flat} \ t2)} \)

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

\( \text{flat (mapTree \ f \ (\text{Node} \ x \ t1 \ t2))} \)
\textbf{Proof} by structural induction on $t$

\textbf{Induction step:}

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

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

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

$\text{flat (mapTree } f (\text{Node } x \ t_1 \ t_2))$

$= \text{flat (Node } (f \ x) \ (\text{mapTree } f \ t_1) \ (\text{mapTree } f \ t_2))$

$= \text{flat (mapTree } f \ t_1) ++ [f \ x] ++ \text{flat (mapTree } f \ t_2)$

---

\textbf{The general (regular) case}

\textbf{data} T a = ...
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: \( \text{flat} \ (\text{mapTree} \ f \ (\text{Node} \ x \ t1 \ t2)) = \text{map} \ f \ (\text{flat} \ (\text{Node} \ x \ t1 \ t2)) \)

\[ \begin{align*}
\text{flat} \ (\text{mapTree} \ f \ (\text{Node} \ x \ t1 \ t2)) \\
= \text{flat} \ (\text{Node} \ (f \ x) \ (\text{mapTree} \ f \ t1) \ (\text{mapTree} \ f \ t2)) \\
= \text{flat} \ (\text{mapTree} \ f \ t1) \ ++ \ [f \ x] \ ++ \ \text{flat} \ (\text{mapTree} \ f \ t2) \\
= \text{map} \ f \ (\text{flat} \ t1) \ ++ \ [f \ x] \ ++ \ \text{map} \ f \ (\text{flat} \ t2) \\
\quad \quad \quad \text{-- by IH1 and IH2}
\end{align*} \]

\[ \text{map} \ f \ (\text{flat} \ (\text{Node} \ x \ t1 \ t2)) \\
= \text{map} \ f \ (\text{flat} \ t1 \ ++ \ [x] \ ++ \ \text{flat} \ t2) \\
= \text{map} \ f \ (\text{flat} \ t1) \ ++ \ [f \ x] \ ++ \ \text{map} \ f \ (\text{flat} \ t2) \\
\quad \quad \quad \text{-- by lemma distributivity of map over ++}
\]

Note: Base case and -- by def of ... omitted

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 \( T \)
Structural induction for Tree

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

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 \( P(C_i \cdot x_1 \ldots x_n) \)
The general (regular) case

data T a = ...

Assumption: T is a **regular** data type:
- Each constructor $C_i$ of T must have a type
  $t_1 \rightarrow \ldots \rightarrow t_n \rightarrow T \ a$
  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 $P(C_i \ x_1 \ldots x_n)$
  assuming the induction hypotheses $P(x_j)$ for all $j$ s.t. $t_j = T \ a$

Example of non-regular type: data $T = C \ [\ T \ ]$

The problem

- So far, only batch programs:

- Haskell programs are pure mathematical functions:
  Haskell programs have no side effects
The problem

- Haskell programs are pure mathematical functions:
  Haskell programs have no side effects
- Reading and writing are 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

```
inputInt :: Int
```
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

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

Now all functions potentially perform side effects.

Now we can no longer reason about Haskell like in mathematics:

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

\[
\text{inputInt} + \text{inputInt} = 2\times\text{inputInt}
\]

... are no longer true.

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

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

Now all functions potentially perform side effects.

Now we can no longer reason about Haskell like in mathematics:

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

\[
\text{inputInt} + \text{inputInt} = 2\times\text{inputInt}
\]

... are no longer true.

The pure solution

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

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

Basic actions

- `getChar :: IO Char`
  Reads a Char from standard input, echoes it to standard output, and returns it as the result.
- `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
```

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

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

Sequencing: do

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

Example

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

General format (observe layout!):

\[
do \ a_1 \\
   \vdots \\
   \ a_n
\]

Sequencing: `do`

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

Example

\[get2 :: IO (Char,Char)\]
\[get2 = \text{do } x \leftarrow \text{getChar} \quad \text{-- result is named } x\]
\[\quad \text{getChar} \quad \text{-- result is ignored}\]
\[\quad y \leftarrow \text{getChar} \]
\[\quad \text{return } (x,y)\]
General format (observe layout!):

do  $a_1$
   :
    $a_n$

where each $a_i$ can be one of

- an action
  Effect: execute action

- $x \leftarrow action$
  Effect: execute $action :: IO a$, give result the name $x :: a$

- let $x = expr$
  Effect: give $expr$ the name $x$
  Lazy: $expr$ is only evaluated when $x$ is needed!

General format (observe layout!):

do  $a_1$
   :
    $a_n$

where each $a_i$ can be one of

- an action
  Effect: execute action

- $x \leftarrow action$
  Effect: execute $action :: IO a$, give result the name $x :: a$

- let $x = expr$
  Effect: give $expr$ the name $x$

Derived primitives

Write a string to standard output:

$putStr :: String \rightarrow IO ()$
Read a line from standard input:

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

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)
```

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:

```haskell
strLen :: IO ()
strLen = do putStrLn "Enter a string: "
            xs <- getLine
            putStrLn "The string has "
            putStrLn
```
Prompt for a string and display its length:

```haskell
strLen :: IO ()
strLen = do putStrLn "Enter a string: "
            xs <- getLine
            putStrLn "The string has "
            putStrLn (show (length xs))
            putStrLn "characters"
>
strLen
```

Enter a string: abc

The string has 3 characters
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`.
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
```

Case study

The game of Hangman
in file hangman.hs