Script generated by TTT

Title: Nipkow: Info2 (10.12.2013)

Date: Tue Dec 10 15:30:48 CET 2013

Duration: 82:29 min

Pages: 149

TTT Tue 15:30 Tobias Nipkow

.3 Case study: boolean formulas

.3 Case study: boolean formulas

```
type Name = String
```

.3 Case study: boolean formulas

```
type Name = String

data Form = F | T
```

.3 Case study: boolean formulas

```
type Name = String

data Form = F | T
          | Var Name
```

.3 Case study: boolean formulas

```
type Name = String

data Form = F | T
          | Var Name
          | Not Form
```

.3 Case study: boolean formulas

```
type Name = String

data Form = F | T
          | Var Name
          | Not Form
          | And Form Form
          | Or Form Form
```

.3 Case study: boolean formulas

```
type Name = String

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

.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"))`

.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
```

.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
```


Now: `Var "p" :|: Not(Var "p")`

Pretty printing

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

Pretty printing

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

instance Show Form where
```

Pretty printing

```
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)
```

Pretty printing

```
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)
```

Pretty printing

```
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)
```

Pretty printing

```
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)
```

Syntax versus meaning

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

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
```

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"` ?

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

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

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

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

eval :: Valuation -> Form -> Bool
eval _ F = False
```

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

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
```


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

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
```


```
eval v (Var x) = the(lookup x v)
```


```
-- 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
```


```
-- 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)
```

```
-- 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)
eval 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)
eval v (p :&: q) = eval v p && eval v q
eval 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)
eval v (p :&: q) = eval v p && eval v q
eval 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]
```

All valuations for a given list of variable names:

```
vals :: [Name] -> [Valuation]
vals [] = [[]]
```

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 ]
```

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:

```
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):[]]
```

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)]
```

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:

```
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"]]
```

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)]
```

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?

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
```

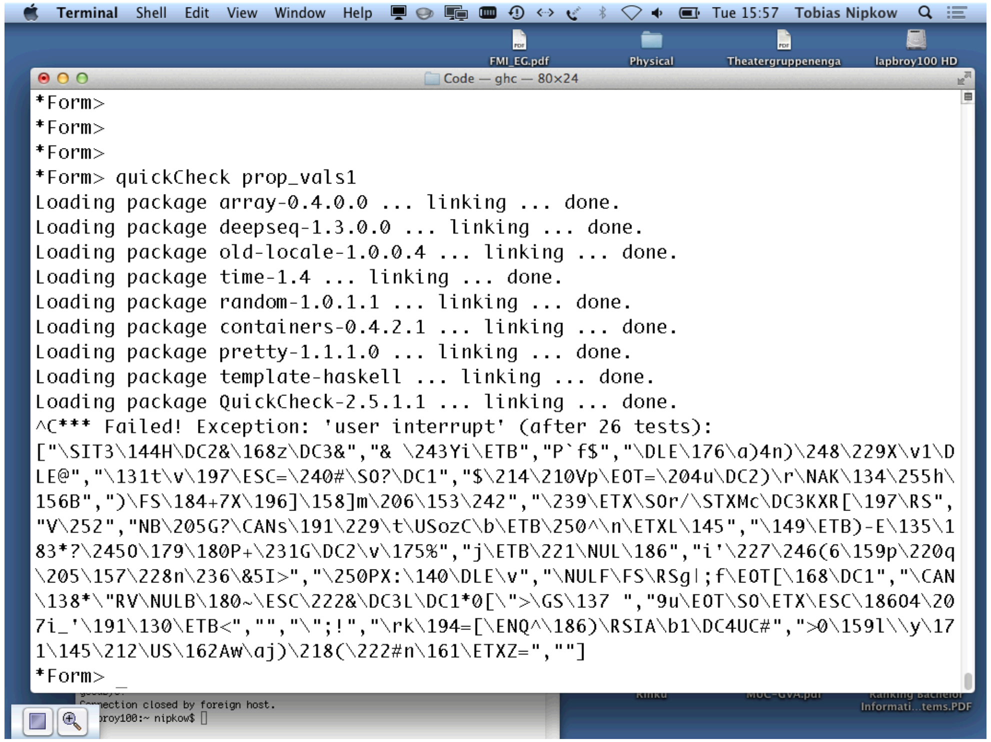

Does `vals` construct *all* valuations?

```
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
```

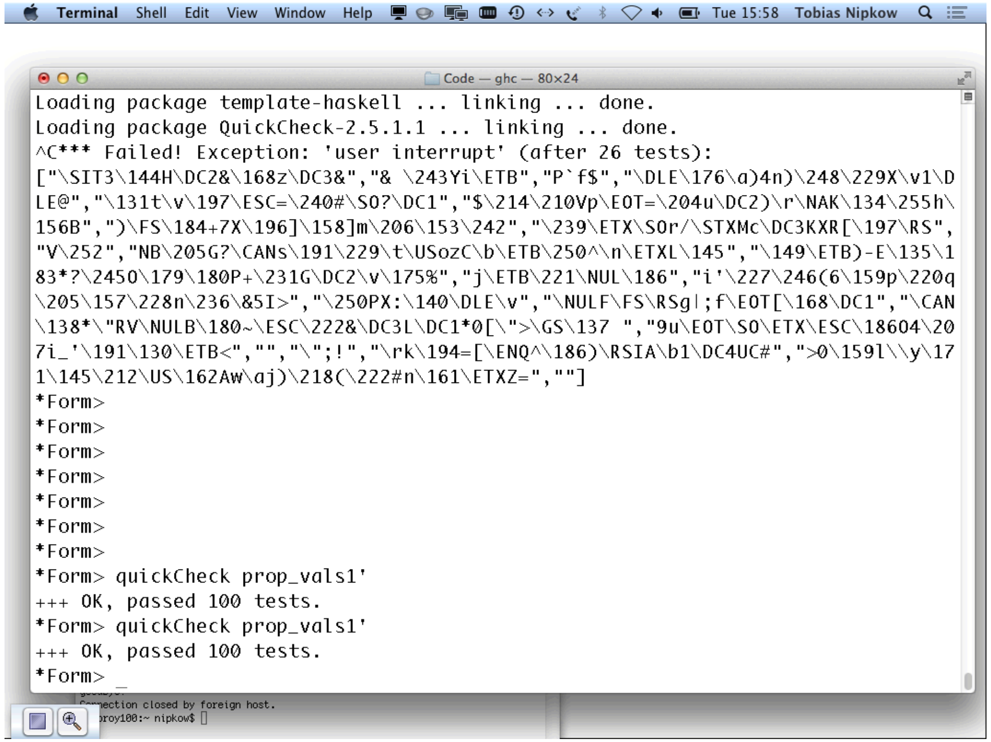

Demo

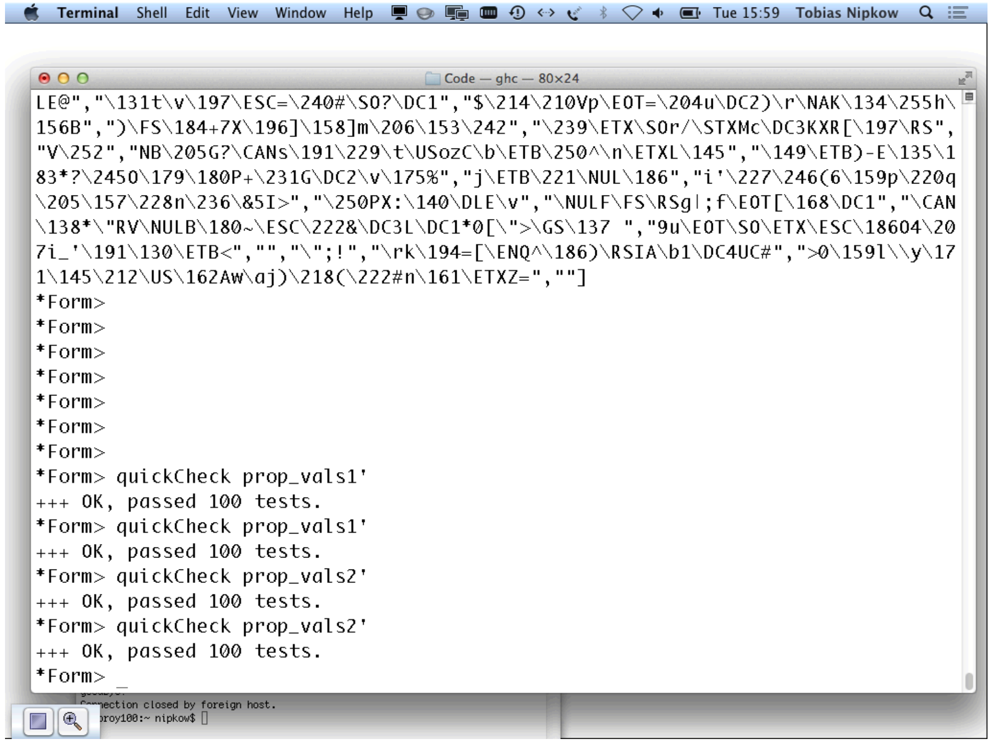


Restrict size of test cases:

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

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



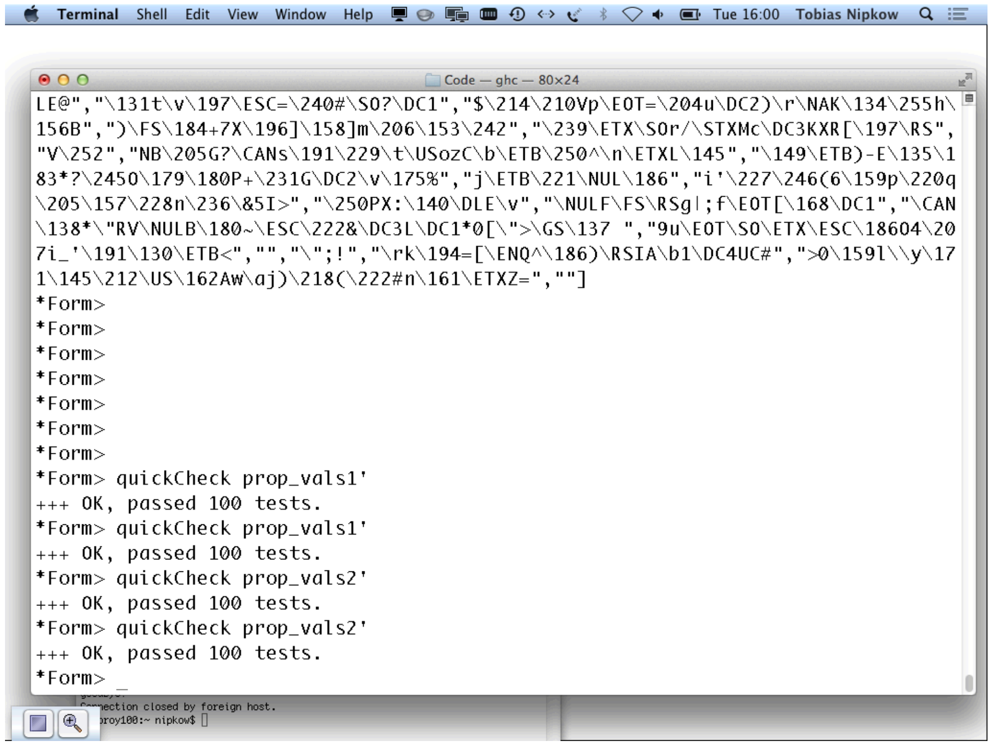


Restrict size of test cases:

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

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

Demo

Satisfiable and tautology

```
satisfiable :: Form -> Bool
```

Satisfiable and tautology

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

Satisfiable and tautology

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

tautology :: Form -> Bool
tautology = not . satisfiable . 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)
```

```
p0 :: Form
p0 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))
```

Simplifying a formula: Not inside?

Simplifying a formula: Not inside?

```
isSimple :: Form -> Bool
```

Simplifying a formula: Not inside?

```
isSimple :: Form -> Bool
isSimple (Not p)     =  not (isOp p)
```

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
```

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
```

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)
```

Simplifying a formula: Not inside!

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

Simplifying a formula: Not inside!

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

Simplifying a formula: Not inside!

```
simplify :: Form -> Form
simplify (Not p)     =  pushNot (simplify p)
  where
  pushNot (Not p)     =  p
  pushNot (p :&: q)   =
```

Simplifying a formula: Not inside!

```
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
```

Simplifying a formula: Not inside!

```
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           =
```

Simplifying a formula: Not inside!

```
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 :: 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
simplify (p :&: q)  =  simplify q :&: simplify q
simplify (p :|: q)  =  simplify p :|: simplify q
```

Simplifying a formula: Not inside!

```
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
simplify (p :&: q)  =  simplify q :&: simplify q
simplify (p :|: q)  =  simplify p :|: simplify q
simplify p          =
```

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!

```
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
simplify (p :&: q)   =  simplify q :&: simplify q
simplify (p :|: q)   =  simplify p :|: simplify q
simplify p           =  p
```

8.4 Structural induction

Structural induction for Tree

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

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)`.

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)`.
(`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)
```

Lemma `flat (mapTree f t) = map f (flat t)`

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

```
flat (mapTree f (Node x t1 t2))
= flat (Node (f x) (mapTree f t1) (mapTree f t2))
```

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

```
flat (mapTree f (Node x t1 t2))
= flat (Node (f x) (mapTree f t1) (mapTree f t2))
```


```
= flat (mapTree f t1) ++ [f x] ++ flat (mapTree f t2)
```

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

```
flat (mapTree f (Node x t1 t2))
= flat (Node (f x) (mapTree f t1) (mapTree f t2))
= flat (mapTree f t1) ++ [f x] ++ flat (mapTree f t2)
= map f (flat t1) ++ [f x] ++ map f (flat t2)
```

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

```
flat (mapTree f (Node x t1 t2))
= flat (Node (f x) (mapTree f t1) (mapTree f t2))
= flat (mapTree f t1) ++ [f x] ++ flat (mapTree f t2)
= map f (flat t1) ++ [f x] ++ map f (flat t2)
    -- by IH1 and IH2
map f (flat (Node x t1 t2))
= map f (flat t1 ++ [x] ++ flat t2)
```

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

```
flat (mapTree f (Node x t1 t2))
= flat (Node (f x) (mapTree f t1) (mapTree f t2))
= flat (mapTree f t1) ++ [f x] ++ flat (mapTree f t2)
= map f (flat t1) ++ [f x] ++ map f (flat t2)
    -- by IH1 and IH2
map f (flat (Node x t1 t2))
= map f (flat t1 ++ [x] ++ flat t2)
= map f (flat t1) ++ [f x] ++ map f (flat t2)
```

The general (regular) case

```
data T a = ...
```

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 `->` `...` `->` t_{n_i} `->` `T a`
such that each t_j is either `T a` or does not contain `T`

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 `->` `...` `->` t_{n_i} `->` `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 $P(C_i\ x_1 \ldots x_{n_i})$

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 `->` `...` `->` t_{n_i} `->` `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 $P(C_i\ x_1 \ldots x_{n_i})$
assuming the induction hypotheses $P(x_j)$ for all j s.t. $t_j =$ `T 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 `->` `...` `->` t_{n_i} `->` `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 $P(C_i\ x_1 \ldots x_{n_i})$

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.

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

```
inputInt :: Int
```

Now all functions potentially perform 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
```

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

```
inputInt :: Int
```

Now all functions potentially perform side effects.

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

```
inputInt - inputInt   =   0
```

The pure solution

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

The pure solution

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.

The pure solution

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`

The pure solution

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`

The pure solution

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).

()

- Type `()` is the type of empty tuples (no fields).
- The only value of type () is `()`, the empty tuple.

()

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

Basic actions

- `getChar :: IO Char`

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 ()`

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`

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  ?
```

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

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

General format (observe layout!):

do a_1
$\vdots$
a_n

General format (observe layout!):

do a_1
$\vdots$
a_n

where each a_i can be one of

General format (observe layout!):

do a_1
$\vdots$
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
$\vdots$
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*
  Effect: give *expr* the name x
  Lazy: *expr* is only evaluated when x is needed!

Derived primitives

Write a string to standard output:

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

Derived primitives

Write a string to standard output:

```
putStr :: String -> IO ()
putStr []       =  return ()
```

Derived primitives

Write a string to standard output:

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

Derived primitives

Write a string to standard output:

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

Write a line to standard output:

```
putStrLn :: IO ()
```

Derived primitives

Write a string to standard output:

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

Write a line to standard output:

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

Read a line from standard input:

```
getLine :: IO String
```

Read a line from standard input:

```
getLine :: IO String
getLine = do x <- getChar
```

Read a line from standard input:

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
```

Read a line from standard input:

```
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:

```
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`.

Derived primitives

Write a string to standard output:

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

Example

Prompt for a string and display its length:

```
strLen :: IO ()
```

Example

Prompt for a string and display its length:

```
strLen :: IO ()
strLen   =  do putStr "Enter a string: "
```

Example

Prompt for a string and display its length:

```
strLen :: IO ()
strLen   =  do putStr "Enter a string: "
               xs <- getLine
```

How to read other types

How to read other types

Input string and convert

How to read other types

Input string and convert

Useful class:

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

How to read other types

Input string and convert

Useful class:

```
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:

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

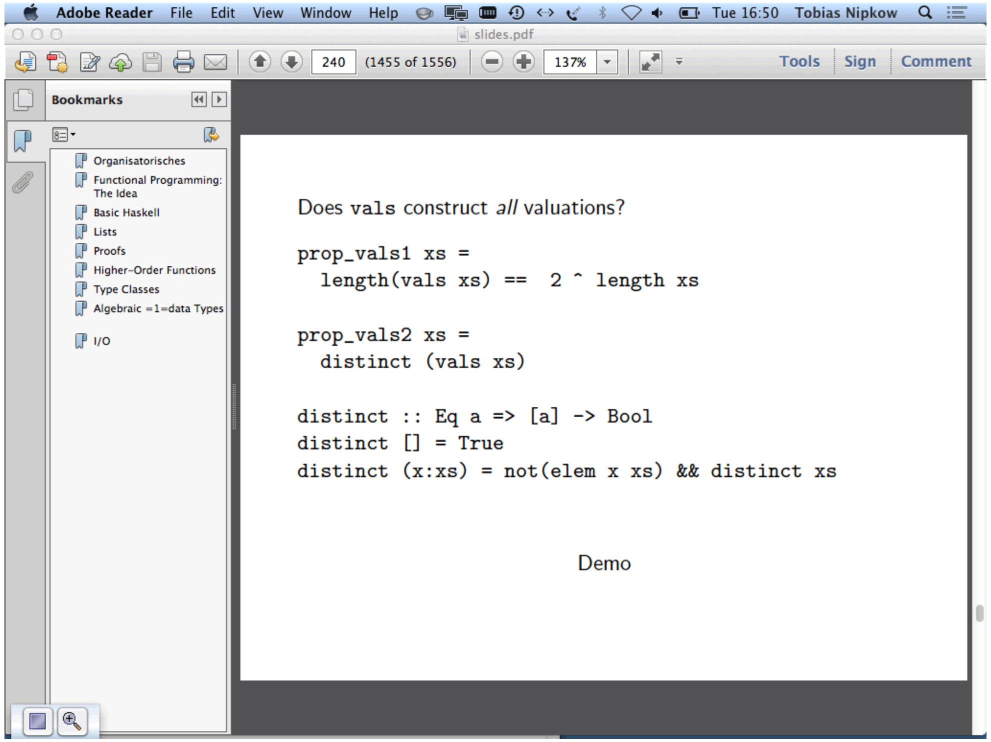
Most predefined types are in class Read.

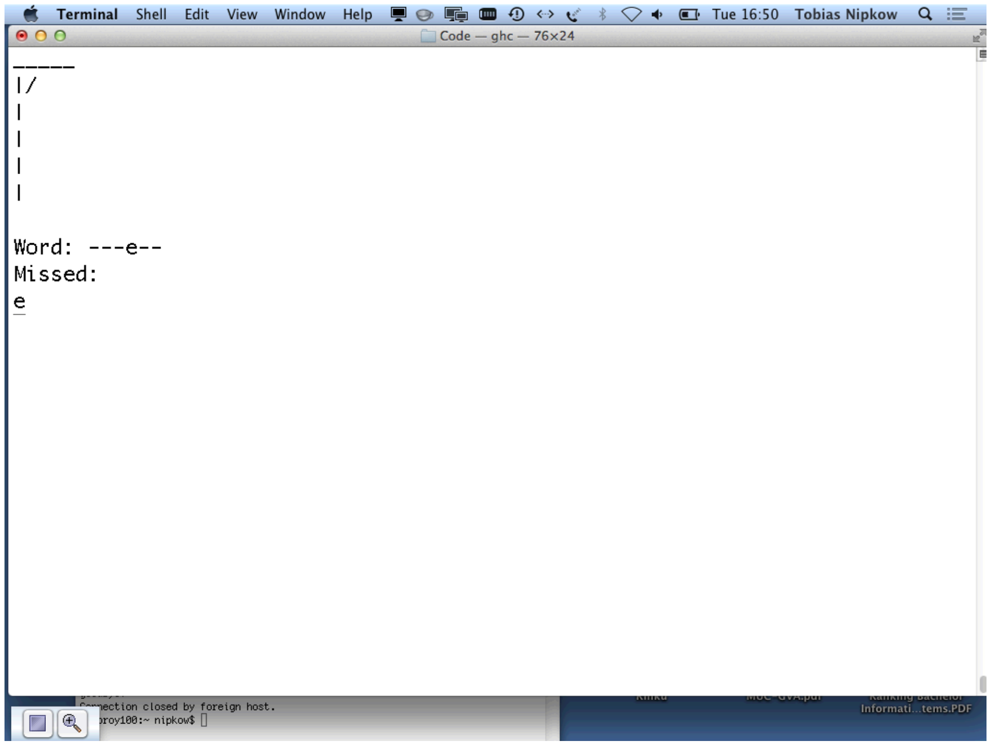
Example:

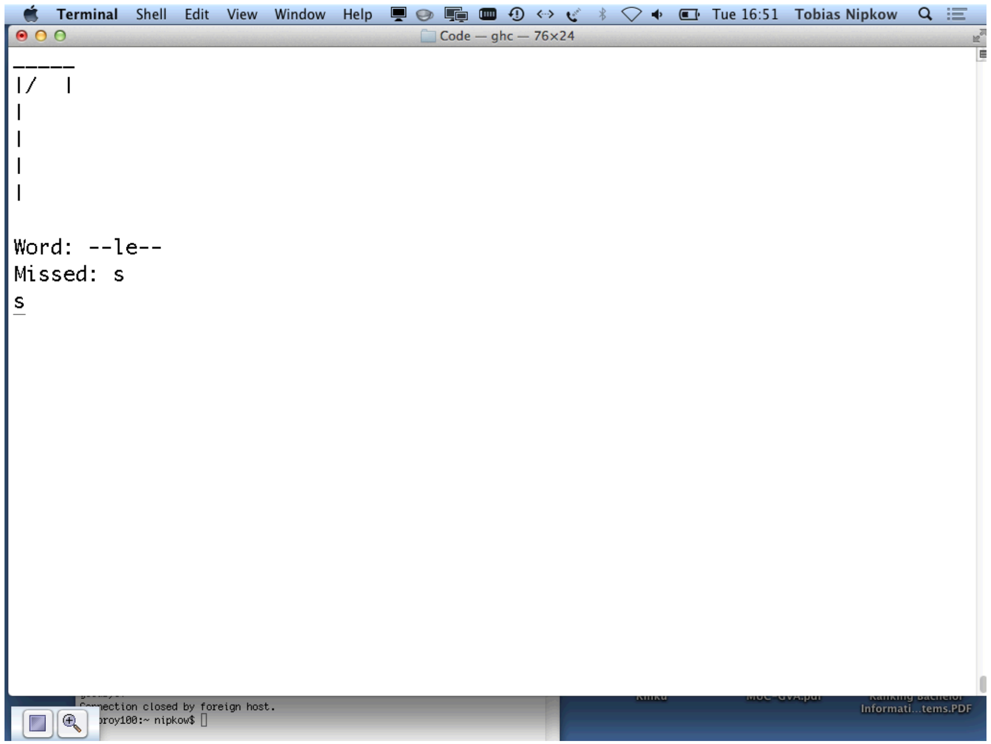
```
getInt :: IO Integer
getInt = do xs <- getLine
            return (read xs)
```

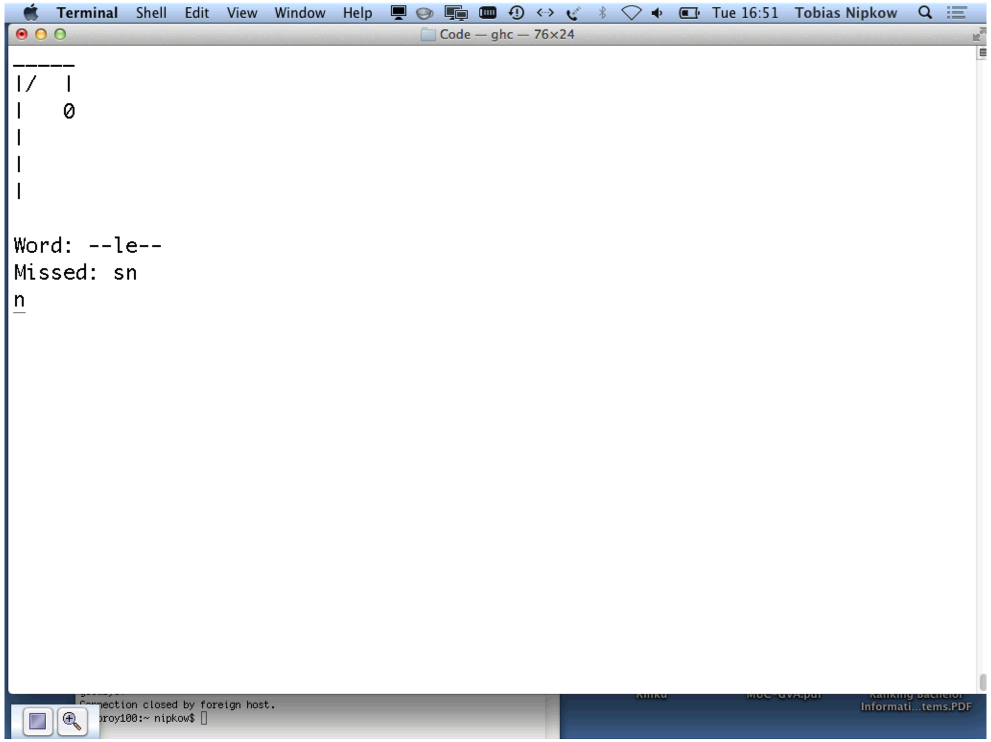
Case study

The game of Hangman
in file `hangman.hs`

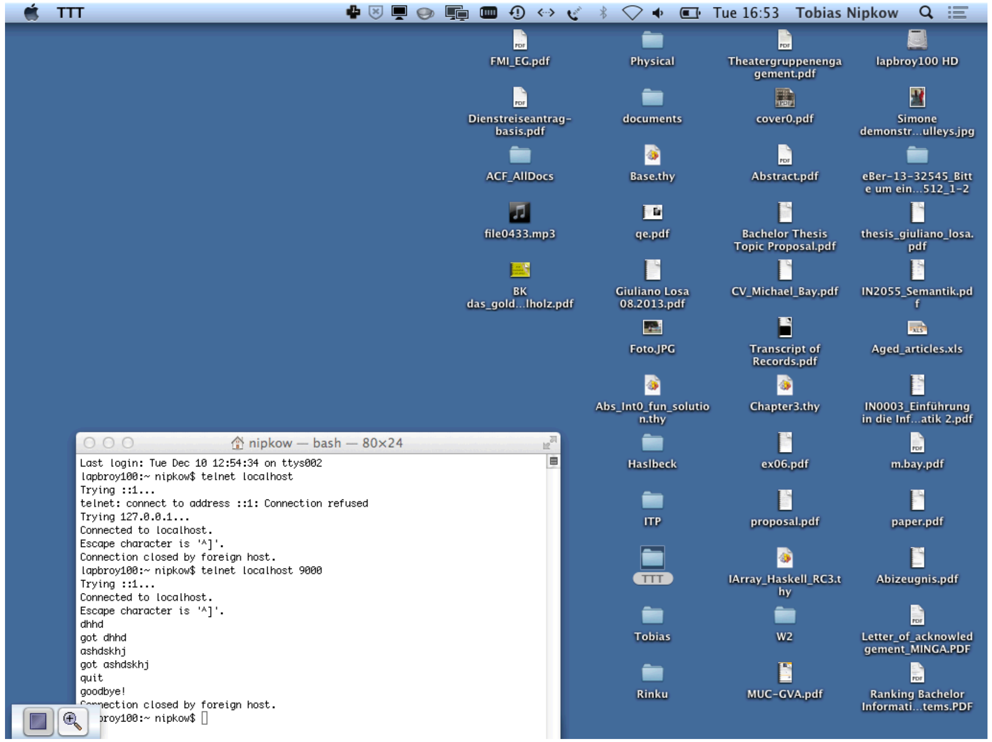








Terminal Shell Edit View Window Help Tue 16:51 Tobias Nipkow
Code — ghc — 76×24
_____
|/ |
| 0
|
|
|
Word: --le--
Missed: sn
n
Connection closed by foreign host.
lapbroy100:~ nipkow$



TTT Tue 16:53 Tobias Nipkow
nipkow — bash — 80×24
Last login: Tue Dec 10 12:54:34 on ttys002
lapbroy100:~ nipkow$ telnet localhost
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
lapbroy100:~ nipkow$ telnet localhost 9000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
dhhd
got dhhd
ashdskhj
got ashdskhj
quit
goodbye!
Connection closed by foreign host.
lapbroy100:~ nipkow$