

Hoare Logic and Propositional Logic

Today

- Function composition
- Assignment statements in functional programming
- While (pre-test) loops in Haskell
- Propositional logic in Haskell
 - Properties of formulas
 - A parser
 - Conjunctive normal form

Recap: Function Composition

Normal order:

$(.) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b)$
 $g . f = \lambda x \rightarrow g (f x)$

Flipped order:

$(\#) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$
 $(\#) = \text{flip } (.)$
 $f \# g = g . f$

Function composition:

If $\{p\} f \{q\}$ and $\{q\} g \{r\}$, then $\{p\} f \# g \{r\}$

Function application, flipped order

```
*Lecture3> :t ($)  
($) :: (a -> b) -> a -> b
```

Flipped:

```
infixl 1 $$  
($$) :: a -> (a -> b) -> b  
($$) = flip ($)
```

```
*Lecture3> 7 $$ succ  
8
```

No Assignment in Pure Functional Programming

$\lambda x \mapsto x+1$ and $x := x+1$, what is the difference?

No Assignment in Pure Functional Programming

$\lambda x \mapsto x+1$ and $x := x+1$, what is the difference?

A: different types,

$\lambda x \mapsto x+1$ is a function

$x := x+1$ is an assignment in the context of memory allocation (an environment)

In case of integers, such an environment could be a function of type

$V \rightarrow \text{Int}$, where V is a set of variables.

Assignment as Update

Updating the definition of a function:

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (x,y) = \ z -> if x == z then y else f z
```

```
updates :: Eq a => (a -> b) -> [(a,b)] -> a -> b
updates = foldl update
```

Examples:

```
>updates succ [(0,0),(100,100)] 4
>updates succ [(0,0),(100,100)] 100
```

```
>updates fromEnum [('a',0),('b',100)] 'a'
>updates fromEnum [('a',0),('b',100)] 'c'
```

What is the outcome?

Assignment as Update

Updating the definition of a function:

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (x,y) = \ z -> if x == z then y else f z
```

```
updates :: Eq a => (a -> b) -> [(a,b)] -> a -> b
updates = foldl update
```

Examples:

```
*Lecture3> updates succ [(0,0),(100,100)] 4
5
```

```
*Lecture3> updates succ [(0,0),(100,100)] 100
100
```

```
*Lecture3> updates fromEnum [('a',0),('b',100)] 'a'
0
```

```
*Lecture3> updates fromEnum [('a',0),('b',100)] 'c'
99
```


Update operation on an environment

To implement variable assignment we need a datatype for expressions

- the assign command assigns an expression to a variable.

```
type Var = String
```

```
type Env = Var -> Integer
```

```
data Expr = I Integer
```

```
        | V Var
```

```
        | Add Expr Expr
```

```
        | Subtr Expr Expr
```

```
        | Mult Expr Expr
```

```
        deriving (Eq,Show)
```

The environment is a finite object => defined only for a finite list of variables

- Initially, it is everywhere undefined

```
initEnv :: Env
```

```
initEnv = \_ -> undefined
```

which can also be expressed as

```
initE = const undefined
```

Operating in an environment

Evaluation of an expression

`eval :: Expr -> Env -> Integer`

`eval (I i) _ = i`

`eval (V name) env = env name`

`eval (Add e1 e2) env = (eval e1 env) + (eval e2 env)`

`eval (Subtr e1 e2) env = (eval e1 env) - (eval e2 env)`

`eval (Mult e1 e2) env = (eval e1 env) * (eval e2 env)`

Variable assignment

`assign :: Var -> Expr -> Env -> Env`

`assign var expr env = update env (var, eval expr env)`

Example

initialize an environment;

$x := 3;$

$y := 5;$

$x := x * y;$

evaluate x

example = initEnv \$\$

assign "x" (I 3) #

assign "y" (I 5) #

assign "x" (Mult (V "x") (V "y")) #

eval (V "x")

Revisit while loops

lather; rinse; repeat

- What is missing here?

A stop condition

until clean (lather # rinse)

Or

While (not . clean) (lather # rinse)

until is predefined in Haskell

How could we define while as a Haskell function?

A while definition

`until :: (a -> Bool) -> (a -> a) -> a -> a`

`until p f x = if p x then x else until p f (f x)`

`while :: (a -> Bool) -> (a -> a) -> a -> a`

`while = until . (not .)`

→ limitation: it assumes x is not in itself a function

Example: Euclid greatest common divisor

```
euclid m n = (m,n) $$  
  while (\ (x,y) -> x /= y)  
    (\ (x,y) -> if x > y then (x-y,y)  
              else (x,y-x)) #  
  fst
```

```
euclid' m n = fst $ eucl (m,n) where  
  eucl = until (uncurry (==))  
    (\ (x,y) -> if x > y then (x-y,y) else (x,y-x))
```

```
uncurry :: (a -> b -> c) -> (a, b) -> c  
curry :: ((a, b) -> c) -> a -> b -> c
```

```
f x y = g (x,y)  
uncurry f = g  
curry g = f
```

While + Return

Suppose we would want to further transform the result:

```
whiler :: (a -> Bool) -> (a -> a) -> (a -> b) -> a -> b  
whiler p f r = while p f # r
```

→ here the variable is a function, r

Now euclid becomes:

```
euclid2 m n = (m,n) $$  
  whiler (\ (x,y) -> x /= y)  
    (\ (x,y) -> if x > y then (x-y,y) else (x,y-x))  
    fst
```

Fibonacci numbers, *functional imperative* style

```
fibonacci :: Integer -> Integer
fibonacci n = fibon (0,1,n) where
  fibon = whiler
    (\ (_,_,n) -> n > 0)
    (\ (x,y,n) -> (y,x+y,n-1))
    (\ (x,_,_) -> x)
```

```
fb :: Integer -> Integer
fb n = fb' 0 1 n where
  fb' x y 0 = x
  fb' x y n = fb' y (x+y) (n-1)
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```


Fibonacci numbers, several styles

Hoare:

```
fibonacci :: Integer -> Integer
fibonacci n = fibon (0,1,n) where
  fibon = whiler
    (\ (_,_,n) -> n > 0)
    (\ (x,y,n) -> (y,x+y,n-1))
    (\ (x,_,_) -> x)
```

Induction:

```
fb :: Integer -> Integer
fb n = fb' 0 1 n where
  fb' x y 0 = x
  fb' x y n = fb' y (x+y) (n-1)
```

Functional:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
or:
fibis = 0 : 1 : zipWith (+) (tail fibis) fibis
```

Boolean Conditions: Statements in propositional logic

Datatype for propositional formulas:

```
type Name = Int
```

```
data Form = Prop Name
          | Neg  Form
          | Cnj  [Form]
          | Dsj  [Form]
          | Impl Form Form
          | Equiv Form Form
          deriving (Eq,Ord)
```

Examples:

```
p = Prop 1
```

```
q = Prop 2
```

```
r = Prop 3
```

```
form1 = Equiv (Impl p q) (Impl (Neg q) (Neg p))
```

```
form2 = Equiv (Impl p q) (Impl (Neg p) (Neg q))
```

```
form3 = Impl (Cnj [Impl p q, Impl q r]) (Impl p r)
```

Show Form

We will define our own show function for formulas:

```
instance Show Form where
  show (Prop x)  = show x
  show (Neg f)   = '-' : show f
  show (Cnj fs)  = "*" ++ showLst fs ++ ")"
  show (Dsj fs)  = "+" ++ showLst fs ++ ")"
  show (Impl f1 f2) = "(" ++ show f1 ++ "=>"
                    ++ show f2 ++ ")"
  show (Equiv f1 f2) = "(" ++ show f1 ++ "<=>"
                    ++ show f2 ++ ")"
```

```
showLst, showRest :: [Form] -> String
showLst [] = ""
showLst (f:fs) = show f ++ showRest fs
showRest [] = ""
showRest (f:fs) = ' ': show f ++ showRest fs
```

Operations on Formulas

Extract proposition symbols:

```
propNames :: Form -> [Name]
propNames = sort.nub.pnames where
  pnames (Prop name) = [name]
  pnames (Neg f)     = pnames f
  pnames (Cnj fs)    = concatMap pnames fs
  pnames (Dsj fs)    = concatMap pnames fs
  pnames (Impl f1 f2) = concatMap pnames [f1,f2]
  pnames (Equiv f1 f2) = concatMap pnames [f1,f2]
```

```
allVals :: Form -> [Valuation]
allVals = genVals . propNames
```

If `propNames` has length n , what is the length of `allVals`?

Generate valuations for proposition symbols:

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

```
genVals :: [Name] -> [Valuation]
genVals [] = [[]]
genVals (name:names) =
  map ((name,True) :) (genVals names)
  ++ map ((name,False) :) (genVals names)
```

Valuations as Environments with Boolean values

type ValFct = Name -> Bool

```
val2fct :: Valuation -> ValFct
val2fct = updates (\_ -> undefined)
```

```
fct2val :: [Name] -> ValFct -> Valuation
fct2val domain f = map (\x -> (x,f x)) domain
```

```
evl :: Valuation -> Form -> Bool
evl [] (Prop c)    = error ("no info: " ++ show c)
evl ((i,b):xs) (Prop c)
  | c == i    = b
  | otherwise = evl xs (Prop c)
evl xs (Neg f)  = not (evl xs f)
evl xs (Cnj fs) = all (evl xs) fs
evl xs (Dsj fs) = any (evl xs) fs
evl xs (Impl f1 f2) = evl xs f1 --> evl xs f2
evl xs (Equiv f1 f2) = evl xs f1 == evl xs f2
```

Satisfiability, Logical Entailment, Equivalence

Formula f is satisfiable if some valuation makes it true:

```
satisfiable :: Form -> Bool  
satisfiable f = any (\ v -> evl v f) (allVals f)
```

A formula f is a tautology if it is satisfied by all valuations.

logicalEntailment - B logically entails A is true if and only if all the valuations that satisfy B also satisfy A .

equivalence - A and B are equivalent

Parsing Propositional Formulas

```
data Form = Prop Name
  | Neg Form
  | Cnj [Form]
  | Dsj [Form]
  | Impl Form Form
  | Equiv Form Form
  deriving (Eq,Ord)
```

```
*Lecture3> parse "(1 +(2 -3))"
["(1 +(2 -3))"]
```

```
*Lecture3> parse "(1 +(2 -3))"
[]
```

```
*Lecture3> parse "(1 +(2 -3)))"
["(1 +(2 -3))"]
```

```
*Lecture3> parseForm (lexer "(1 +(2 -3)))")
["(1 +(2 -3))",[TokenCP,TokenCP]]
```

Parser for token type a that constructs datatype b

`type Parser a b = [a] -> [(b,[a])]`

- Remainder list in result contains tokens not used in the construction of the datatype
- If output list is empty, parsing has not succeeded
- If the output list has more than 1 element, the input was ambiguous

Simplest parser:

`succeed :: b -> Parser a b`
`succeed x xs = [(x,xs)]`

Parser implementation in your lab:

`parseForm :: Parser Token Form`

What is the input of the `parseForm` function?

Conjunctive Normal Form

A **literal** is an atom p or the negation of an atom $\neg p$.

A **clause** is a disjunction of literals.

A formula C is in conjunctive normal form (or: CNF) if it is a conjunction of clauses.

Important for automated theorem proving

- CNF formulas can easily be tested for validity: check that each clause contains some p and its $\neg p$

How do we obtain it?

Step 1: remove arrows

- use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of \rightarrow symbols
- use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$ to get rid of \leftrightarrow symbols

Q: any other equivalence you could use here?

Conjunctive Normal Form

Step 1: remove arrows

- use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of \rightarrow symbols
- use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$ to get rid of \leftrightarrow symbols

Q: any other equivalence you could use here?

A: $p \leftrightarrow q$ and $(p \wedge q) \vee (\neg p \wedge \neg q)$

Q: are there any preconditions for `arrowfree :: Form -> Form` ?

Check `Lecture3.hs` for an `arrowfree` implementation.

Conjunctive Normal Form

Step 1: remove arrows

- use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of \rightarrow symbols
- use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$ to get rid of \leftrightarrow symbols

Q: any other equivalence you could use here?

A: $p \leftrightarrow q$ and $(p \wedge q) \vee (\neg p \wedge \neg q)$

Q: are there any preconditions for arrowfree :: Form \rightarrow Form ?

A: No, it should work for any formula

Q: are there any postconditions for arrowfree?

Conjunctive Normal Form

Step 1: remove arrows

- use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of \rightarrow symbols
- use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$ to get rid of \leftrightarrow symbols

Q: any other equivalence you could use here?

A: $p \leftrightarrow q$ and $(p \wedge q) \vee (\neg p \wedge \neg q)$

Q: are there any preconditions for `arrowfree :: Form -> Form` ?

A: No, it should work for any formula

Q: are there any postconditions for `arrowfree :: Form -> Form`?

A: Yes:

- the result should have no occurrences of `Impl` and `Equiv`.
- The result should be logically equivalent to the original.

CNF (cont'd)

Step 2: conversion to negation normal form (only atoms may be negated)

- Use the equivalence between $\neg\neg\phi$ and ϕ ,
- Use the equivalence between $\neg(\phi \wedge \psi)$ and $\neg\phi \vee \neg\psi$,
- Use the equivalence between $\neg(\phi \vee \psi)$ and $\neg\phi \wedge \neg\psi$

Q: are there any preconditions for `nnf :: Form -> Form`?

Check `Lecture3.hs` for an implementation of `nnf`.

CNF (cont'd)

Step 2: conversion to negation normal form (only atoms may be negated)

- Use the equivalence between $\neg\neg\phi$ and ϕ ,
- Use the equivalence between $\neg(\phi \wedge \psi)$ and $\neg\phi \vee \neg\psi$,
- Use the equivalence between $\neg(\phi \vee \psi)$ and $\neg\phi \wedge \neg\psi$

Q: are there any preconditions for $\text{nnf} :: \text{Form} \rightarrow \text{Form}$?

A: input formula is arrowfree

Q: are there any postconditions for $\text{nnf} :: \text{Form} \rightarrow \text{Form}$?

CNF (cont'd)

Step 2: conversion to negation normal form (only atoms may be negated)

- Use the equivalence between $\neg\neg\phi$ and ϕ ,
- Use the equivalence between $\neg(\phi \wedge \psi)$ and $\neg\phi \vee \neg\psi$,
- Use the equivalence between $\neg(\phi \vee \psi)$ and $\neg\phi \wedge \neg\psi$

Q: are there any preconditions for $\text{nnf} :: \text{Form} \rightarrow \text{Form}$?

A: input formula is arrowfree

Q: are there any postconditions for $\text{nnf} :: \text{Form} \rightarrow \text{Form}$?

A: formula is arrowfree;

only atoms are negated in the formula

Logically equivalent to input formula

Q: what are the pre/postconditions for $\text{nnf}.\text{arrowfree}$?

Parser (cont'd)

Success if we parse a closed parenthesis:

```
parseForms :: Parser Token [Form]
parseForms (TokenCP : tokens) = succeed [] tokens
parseForms tokens =
  [(f:fs, rest) | (f,ys) <- parseForm tokens, (fs,rest) <- parseForms ys ]
```

We parse implications and equivalences differently (these are infix operators):

```
parseImpl :: Parser Token Form
parseImpl (TokenImpl : tokens) =
  [ (f,ys) | (f,y:ys) <- parseForm tokens, y == TokenCP ]
parseImpl tokens = []
```

```
parseEquiv :: Parser Token Form
parseEquiv (TokenEquiv : tokens) =
  [ (f,ys) | (f,y:ys) <- parseForm tokens, y == TokenCP ]
parseEquiv tokens = []
```

Finally:

```
parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]
```