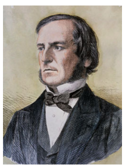
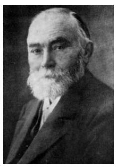
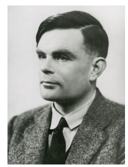
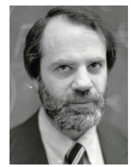
Logic, (Functional) Programming, Model Checking

Jan van Eijck
CWI & ILLC, Amsterdam

Guest Lecture Logic in AI
May 22, 2014

Abstract

This lecture will combine the topics of the title in various ways. First I will show that logic is part of every programming language, in the form of boolean expressions. Next, we will analyze the language of boolean expressions a bit, looking both at syntax and semantics. If the language of boolean expressions is enriched with quantifiers, we move from propositional logic to predicate logic. I will discuss how the expressions of that language can describe the ways things are. Next, I will say something about model checking, and about the reverse side of the expressive power of predicate logic. I will end with a brief sketch of epistemic model checking. In the course of the lecture I will connect everything with (functional) programming, and you will be able to pick up some Haskell as we go along.

Every Programming Language Uses Logic

From a Java Exercise: suppose the value of b is false and the value of x is 0. Determine the value of each of the following expressions:

```
b && x == 0
b || x == 0
!b && x == 0
!b || x == 0
b && x != 0
b || x != 0
!b && x != 0
!b || x != 0
```

Question 1 *What are the answers to the Java exercise?*

The Four Main Ingredients of Imperative Programming

Assignment Put number 123 in location x

Concatenation First do this, next do that

Choice If this condition is true then do this, else do that.

Loop As long as this condition is true, do this.

The conditions link to a language of logical expressions that has negation, conjunction and disjunction.

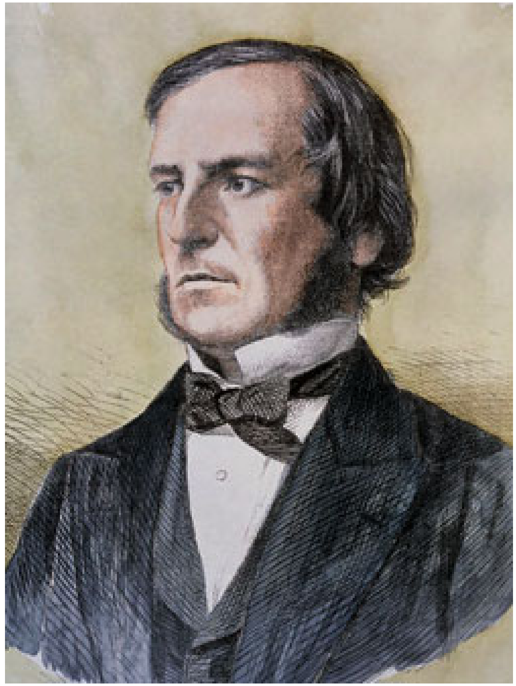
Let's Talk About Logic

Talking about logic means: talking about a logical language.

The language of expressions in programming is called Boolean logic or propositional logic.

Context free grammar for propositional logic:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi.$$



George Boole (1815 – 1864)

Literate Programming, in Haskell

See `http://www.haskell.org`

```
module Logic

where

import Data.List
import Data.Char
```

A Datatype for Formulas

```
type Name = Int

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

This looks almost the same as the grammar for propositional logic.

Example Formulas

```
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)
```

Validity

```
*Logic> form1
((1==>2)<=>(-2==>-1))
*Logic> form2
((1==>2)<=>(-1==>-2))
*Logic> form3
(*((1==>2),(2==>3))==>(1==>3))
```

Question 2 *A formula that is always true, no matter whether its proposition letters are true, is called valid. Which of these three formulas are valid?*

Validity

```
*Logic> form1
((1==>2)<=>(-2==>-1))
*Logic> form2
((1==>2)<=>(-1==>-2))
*Logic> form3
(*((1==>2),(2==>3))==>(1==>3))
```

Question 2 *A formula that is always true, no matter whether its proposition letters are true, is called valid. Which of these three formulas are valid?*

Answer: `form1` and `form3`.

Proposition Letters (Indices) Occurring in a Formula

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

To understand what happens here, we need to learn a bit more about functional programming.

Question 3 *Why is it important to know which proposition letters occur in a formula?*

Proposition Letters (Indices) Occurring in a Formula

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

To understand what happens here, we need to learn a bit more about functional programming.

Question 3 *Why is it important to know which proposition letters occur in a formula?*

Answer: Because the truth of the formula depends on the truth/falsity of these.

Type Declarations

```
propNames :: Form -> [Name]
```

This is a type declaration or type specification. It says that `propNames` is a function that takes a `Form` as an argument, and yields a list of `Names` as a value. `[Name]` is the type of a list of `Names`.

This function is going to give us the names (indices) of all proposition letters that occur in a formula.

map, concat, sort, nub

If you use the command `:t` to find the types of the predefined function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

This tells you that map is a higher order function: a function that takes other functions as arguments. map takes a function of type `a -> b` as its first argument, and yields a function of type `[a] -> [b]` (from lists of `a`s to lists of `b`s).

In fact, the function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

Thus `map pnames fs` is the command to apply the `pnames` function to all members for `fs` (a list of formulas) and collect the results in a new list.

Question 4 *What is the type of this new list?*

map, concat, sort, nub

If you use the command `:t` to find the types of the predefined function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

This tells you that map is a higher order function: a function that takes other functions as arguments. map takes a function of type `a -> b` as its first argument, and yields a function of type `[a] -> [b]` (from lists of `a`s to lists of `b`s).

In fact, the function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

Thus `map pnames fs` is the command to apply the `pnames` function to all members for `fs` (a list of formulas) and collect the results in a new list.

Question 4 *What is the type of this new list?*

Answer: a list of lists of names: `[[Name]]`.

Mapping

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here `(^2)` is a short way to refer to the squaring function.

Mapping

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

`[1, 4, 9, 16, 25, 36, 49, 64, 81]`

Here `(^2)` is a short way to refer to the squaring function.

Here is a definition of `map`, including a type declaration.

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

In Haskell, the colon `:` is used for putting an element in front of a list to form a new list.

Question 5 *What does* `(x:xs)` *mean? Why does* `map` *occur on the righthand-side of the second equation?*

Mapping

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

`[1, 4, 9, 16, 25, 36, 49, 64, 81]`

Here `(^2)` is a short way to refer to the squaring function.

Here is a definition of `map`, including a type declaration.

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

In Haskell, the colon `:` is used for putting an element in front of a list to form a new list.

Question 5 *What does* `(x:xs)` *mean? Why does* `map` *occur on the righthand-side of the second equation?*

Answer: `(x:xs)` is the pattern of a non-empty list. The call on the righthand side of the second equation is an example of recursion.

List Concatenation: ++

++ is the operator for concatenating two lists. Look at the type:

```
*Logic> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

Question 6 *Can you figure out the definition of* ++*?*

List Concatenation: ++

++ is the operator for concatenating two lists. Look at the type:

```
*Logic> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

Question 6 *Can you figure out the definition of* ++*?*

Answer:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

List Concatenation: concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

Question 7 *What does* `concat` *do?*

List Concatenation: concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

Question 7 *What does* `concat` *do?*

Answer: `concat` takes a list of lists and constructs a single list. Example:

```
*Logic> concat [[1,2],[4,5],[7,8]]
[1,2,4,5,7,8]
```

filter and nub

Before we can explain `nub` we must understand `filter`. Here is the type:

```
filter :: (a -> Bool) -> [a] -> [a]
```

`Bool` is the type of a Boolean: True or False. So `a -> Bool` is a property.

Here is an example of how `filter` is used:

```
*Logic> filter even [2,3,5,7,8,9,10]
[2,8,10]
```

Question 8 *Can you figure out the definition of* `filter`*?*

filter and nub

Before we can explain `nub` we must understand `filter`. Here is the type:

```
filter :: (a -> Bool) -> [a] -> [a]
```

`Bool` is the type of a Boolean: True or False. So `a -> Bool` is a property.

Here is an example of how `filter` is used:

```
*Logic> filter even [2,3,5,7,8,9,10]
[2,8,10]
```

Question 8 *Can you figure out the definition of* `filter`*?*

Answer:

```
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs
                         else     filter p xs
```

Removing duplicates from a list with nub

Example of the use of `nub`:

```
*Logic> nub [1,2,3,4,1,2,5]
[1,2,3,4,5]
```

Question 9 *Can you figure out the type of* `nub`*?*

Removing duplicates from a list with nub

Example of the use of `nub`:

```
*Logic> nub [1,2,3,4,1,2,5]
[1,2,3,4,5]
```

Question 9 *Can you figure out the type of* `nub`*?*

Answer: `nub :: Eq a => [a] -> [a]`. The `Eq a` means that equality has to be defined for `a`.

Question 10 *Can you figure out the definition of* `nub`*?*

Removing duplicates from a list with nub

Example of the use of `nub`:

```
*Logic> nub [1,2,3,4,1,2,5]
[1,2,3,4,5]
```

Question 9 *Can you figure out the type of* `nub`*?*

Answer: `nub :: Eq a => [a] -> [a]`. The `Eq a` means that equality has to be defined for `a`.

Question 10 *Can you figure out the definition of* `nub`*?*

Answer:

```
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Sorting a list

In order to sort a list (put their elements in some order), we need to be able to compare their elements for size. This is can be done with `compare`:

```
*Logic> compare 3 4
LT
*Logic> compare 'C' 'B'
GT
*Logic> compare [3] [3]
EQ
*Logic> compare "smart" "smile"
LT
```

In order to define our own sorting algorithm, let's first define a function for inserting an item at the correct position in an ordered list.

Question 11 *Can you figure out how to do that? The type is*

```
myinsert :: Ord a => a -> [a] -> [a].
```

Sorting a list

In order to sort a list (put their elements in some order), we need to be able to compare their elements for size. This is can be done with `compare`:

```
*Logic> compare 3 4
LT
*Logic> compare 'C' 'B'
GT
*Logic> compare [3] [3]
EQ
*Logic> compare "smart" "smile"
LT
```

In order to define our own sorting algorithm, let's first define a function for inserting an item at the correct position in an ordered list.

Question 11 *Can you figure out how to do that? The type is*

```
myinsert :: Ord a => a -> [a] -> [a].
```

Answer:

```
myinsert :: Ord a => a -> [a] -> [a]
myinsert x [] = [x]
myinsert x (y:ys) = if compare x y == GT
                      then y : myinsert x ys
                      else x:y:ys
```

Answer:

```
myinsert :: Ord a => a -> [a] -> [a]
myinsert x [] = [x]
myinsert x (y:ys) = if compare x y == GT
                    then y : myinsert x ys
                    else x:y:ys
```

Question 12 *Can you now implement your own sorting algorithm? The type is* `mysort :: Ord a => [a] -> [a]`.

Answer:

```
myinsert :: Ord a => a -> [a] -> [a]
myinsert x [] = [x]
myinsert x (y:ys) = if compare x y == GT
                    then y : myinsert x ys
                    else x:y:ys
```

Question 12 *Can you now implement your own sorting algorithm? The type is* `mysort :: Ord a => [a] -> [a]`.

```
mysort :: Ord a => [a] -> [a]
mysort [] = []
mysort (x:xs) = myinsert x (mysort xs)
```

Valuations

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

All possible valuations for list of prop letters:

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

All possible valuations for a formula, with function composition:

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

Composing functions with ‘.’

The composition of two functions f and g, pronounced ‘f after g’ is the function that results from first applying g and next f.

Standard notation for this: $f \cdot g$. This is pronounced as “f after g”.

Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Note the types! Note the lambda abstraction.

Lambda Abstraction

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

The standard mathematical notation for this is $\lambda x \mapsto x * x$. Haskell notation aims at remaining close to mathematical notation.

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.
- This is precisely what the type-indication `Int -> Int` expresses.

Blowup

Question 13 *If a propositional formula has 20 variables, how many different valuations are there for that formula?*

Blowup

Question 13 *If a propositional formula has 20 variables, how many different valuations are there for that formula?*

Answer: look at this:

```
*Logic> map (2^) [1..20]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,
 16384,32768,65536,131072,262144,524288,1048576]
```

The number doubles with every extra variable, so it grows exponentially.

Question 14 *Does the definition of* `genVals` *use a feasible algorithm?*

Blowup

Question 13 *If a propositional formula has 20 variables, how many different valuations are there for that formula?*

Answer: look at this:

```
*Logic> map (2^) [1..20]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,
 16384,32768,65536,131072,262144,524288,1048576]
```

The number doubles with every extra variable, so it grows exponentially.

Question 14 *Does the definition of* `genVals` *use a feasible algorithm?*

Answer: no.

Evaluation of Formulas

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

Question 15 *Does the definition of* `eval` *use a feasible algorithm?*

Evaluation of Formulas

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

Question 15 *Does the definition of* `eval` *use a feasible algorithm?*

Answer: yes.

New functions: not, all, any

Let's give our own implementations:

New functions: not, all, any

Let’s give our own implementations:

```
mynot :: Bool -> Bool
mynot True = False
mynot False = True
```

New functions: not, all, any

Let's give our own implementations:

```
mynot :: Bool -> Bool
mynot True = False
mynot False = True
```

```
myall :: Eq a => (a -> Bool) -> [a] -> Bool
myall p [] = True
myall p (x:xs) = p x && myall p xs
```

New functions: not, all, any

Let's give our own implementations:

```
mynot :: Bool -> Bool
mynot True = False
mynot False = True
```

```
myall :: Eq a => (a -> Bool) -> [a] -> Bool
myall p [] = True
myall p (x:xs) = p x && myall p xs
```

```
myany :: Eq a => (a -> Bool) -> [a] -> Bool
myany p [] = False
myany p (x:xs) = p x || myany p xs
```

Satisfiability

A formula is satisfiable if some valuation makes it true.

We know what the valuations of a formula `f` are. These are given by

```
allVals f
```

We also know how to express that a valuation `v` makes a formula `f` true:

```
eval v f
```

This gives:

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

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Answer: no, for the call to `allVals` causes an exponential blowup.

Question 17 *Can you think of a feasible algorithm for* `satisfiable`*?*

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Answer: no, for the call to `allVals` causes an exponential blowup.

Question 17 *Can you think of a feasible algorithm for* `satisfiable`*?*

Answer: not very likely …

Question 18 *Does a feasible algorithm for* `satisfiable` *exist?*

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Answer: no, for the call to `allVals` causes an exponential blowup.

Question 17 *Can you think of a feasible algorithm for* `satisfiable`*?*

Answer: not very likely . . .

Question 18 *Does a feasible algorithm for* `satisfiable` *exist?*

Answer: nobody knows. This is the famous P versus NP problem. If I am allowed to guess a valuation for a formula, then the `eval` check whether the valuation makes the formula true takes a polynomial number of steps in the size of the formula. But I first have to find such a valuation, and the number of candidates is exponential in the size of the formula. All known algorithms for `satisfiable` take an exponential number of steps, in the worst case . . .

List Comprehension

List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

Example:

```
someEvens     = [ x | x <- [1..1000], even x ]
```

Equivalently:

```
someEvens     = filter even [1..1000]
```

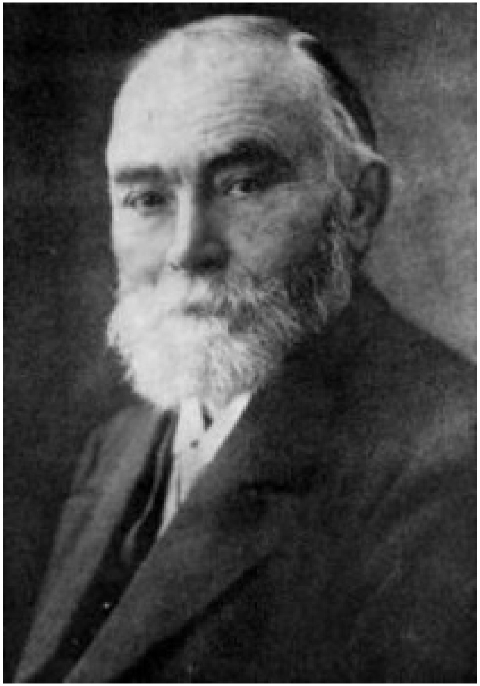
Further Exercises

You are invited to write implementations of:

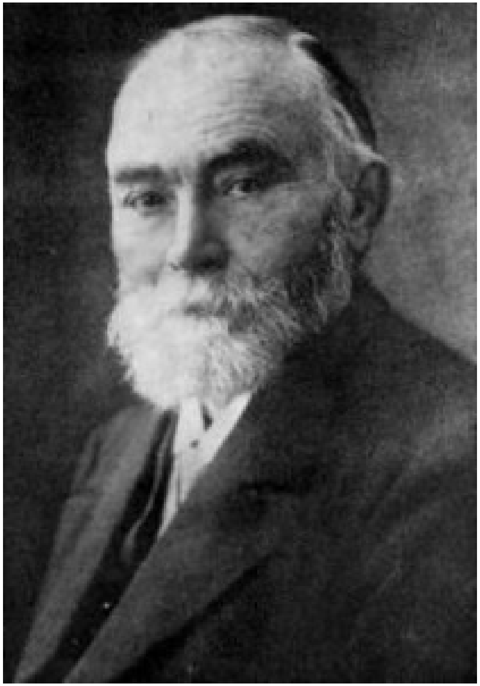
```
contradiction :: Form -> Bool

tautology :: Form -> Bool

-- logical entailment
entails :: Form -> Form -> Bool

-- logical equivalence
equiv :: Form -> Form -> Bool
```

and to test your definitions for correctness.

Who is Who in Logic?



Who is Who in Logic?



Gottlob Frege (1848 – 1925)

Predicate Logic

Assume a set of function symbols is given, and let f range over function symbols.

Assume a set of predicate symbols is given, and let P range over predicate symbols.

$$
\begin{array}{rcl}
t & ::= & x \mid f(t_1, \ldots, t_n) \\
\varphi & ::= & P(t_1, \ldots, t_n) \mid t_1 = t_2 \\
& & \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi) \\
& & \mid (\forall x \varphi) \mid (\exists x \varphi)
\end{array}
$$

Syntax of First Order Logic in Haskell: Terms

```
type Nm = String
data Term = V Nm | F Nm [Term] deriving (Eq,Ord)
```

```
x, y, z :: Term
x = V "x"
y = V "y"
z = V "z"
```

Operations on Terms: Finding the Variables

```
varsInTerm :: Term -> [Nm]
varsInTerm (V name) = [name]
varsInTerm (F _ ts) = varsInTerms ts where
  varsInTerms :: [Term] -> [Nm]
  varsInTerms = nub . concat . map varsInTerm
```

This is similar to finding the names in a propositional formula.

Syntax of First Order Logic in Haskell: Formulas

```
data Formula = Atom Nm [Term]
               | Eq Term Term
               | Ng  Formula
               | Imp Formula Formula
               | Equi Formula Formula
               | Conj [Formula]
               | Disj [Formula]
               | Forall Nm Formula
               | Exists Nm Formula
               deriving (Eq,Ord)
```

```
r0 = Atom "R"

formula1 = Forall "x" (r0 [x,x])
formula2 = Forall "x"
             (Forall "y"
               (Imp (r0 [x,y]) (r0 [y,x])))
formula3 = Forall "x"
             (Forall "y"
              (Forall "z"
               (Imp (Conj [r0 [x,y], r0 [y,z]])
                    (r0 [x,z]))))
```

```
r0 = Atom "R"

formula1 = Forall "x" (r0 [x,x])
formula2 = Forall "x"
             (Forall "y"
               (Imp (r0 [x,y]) (r0 [y,x])))
formula3 = Forall "x"
             (Forall "y"
              (Forall "z"
               (Imp (Conj [r0 [x,y], r0 [y,z]])
                    (r0 [x,z]))))
```

```
*Logic> formula1
A x R[x,x]
*Logic> formula2
A x A y (R[x,y]==>R[y,x])
*Logic> formula3
A x A y A z (*[R[x,y],R[y,z]]==>R[x,z])
```

Who is Who in Logic (2)?

Who is Who in Logic (2)?

Kurt Gödel (1906 – 1978)

Semi-decidability of First Order Predicate Logic

Kurt Gödel showed in his PhD thesis (1929) that a sound and complete proof calculus for first order predicate logic exists.

It follows from this that the notion of logical consequence for predicate logic is semi-decidable. Since proofs are finite, it is possible to enumerate all finite derivations, so there exists a finite enumeration of pairs of first order sentences (φ, ψ) such that ψ is a logical consequence of φ.

Who is Who in Logic (3)?

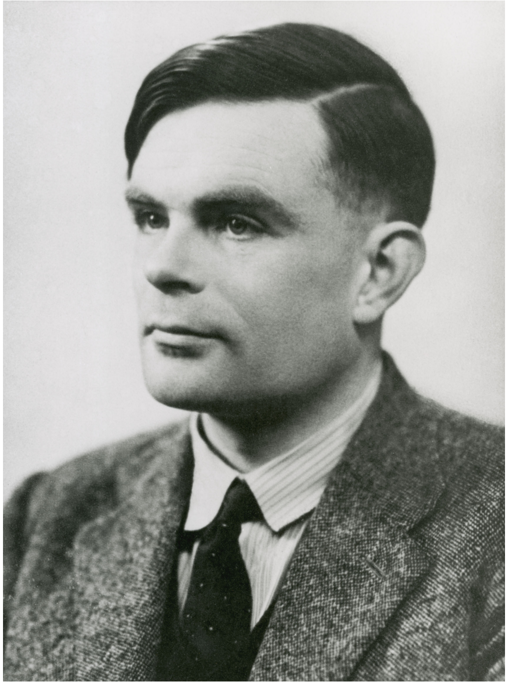
Who is Who in Logic (3)?

Alfred Tarski (1901 – 1983)

Notion of Truth in Formal Languages

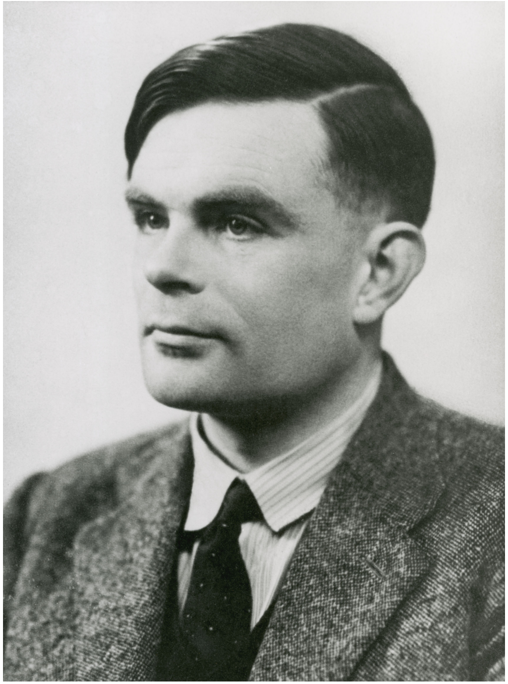
```
evalFOL :: Eq a =>
   [a] -> Lookup a -> Fint a -> Rint a -> Formula -> Bool
evalFOL domain g f i = evalFOL' g where
  evalFOL' g (Atom name ts) = i name (map (termVal g f) ts)
  evalFOL' g (Eq t1 t2) = termVal g f t1 == termVal g f t2
  evalFOL' g (Ng form) = not (evalFOL' g form)
  evalFOL' g (Imp f1 f2) = not
                    (evalFOL' g f1 && not (evalFOL' g f2))
  evalFOL' g (Equi f1 f2) = evalFOL' g f1 == evalFOL' g f2
  evalFOL' g (Conj fs)    = and (map (evalFOL' g) fs)
  evalFOL' g (Disj fs)    = or  (map (evalFOL' g) fs)
  evalFOL' g (Forall v form) =
    all (\ d -> evalFOL' (changeLookup g v d) form) domain
  evalFOL' g (Exists v form) =
    any (\ d -> evalFOL' (changeLookup g v d) form) domain
```

Who is Who in Logic (4)?



Who is Who in Logic (4)?

Alonzo Church (1903 – 1995)



Alan Turing (1912 – 1954)

Limitations of First Order Predicate Logic

Some New Billboards

There are some questions
that can't be answered by logic

There are some questions
that can't be answered
by computing machines

Undecidable Queries

Undecidable Queries

- The deep reason behind the undecidability of first order logic is the fact that its expressive power is so great that it is possible to state undecidable queries.

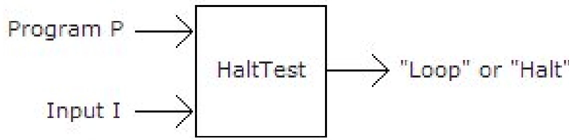
Undecidable Queries

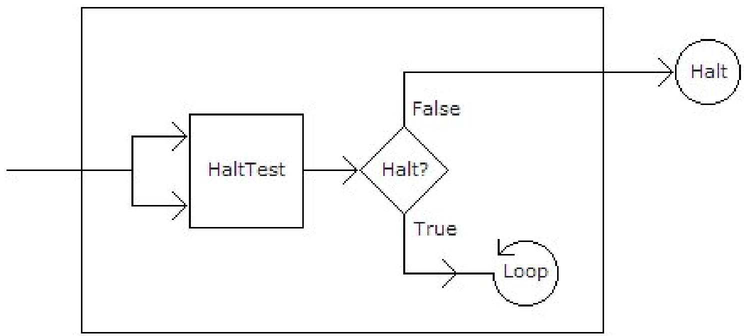
- The deep reason behind the undecidability of first order logic is the fact that its expressive power is so great that it is possible to state undecidable queries.
- One of the famous undecidable queries is the halting problem.

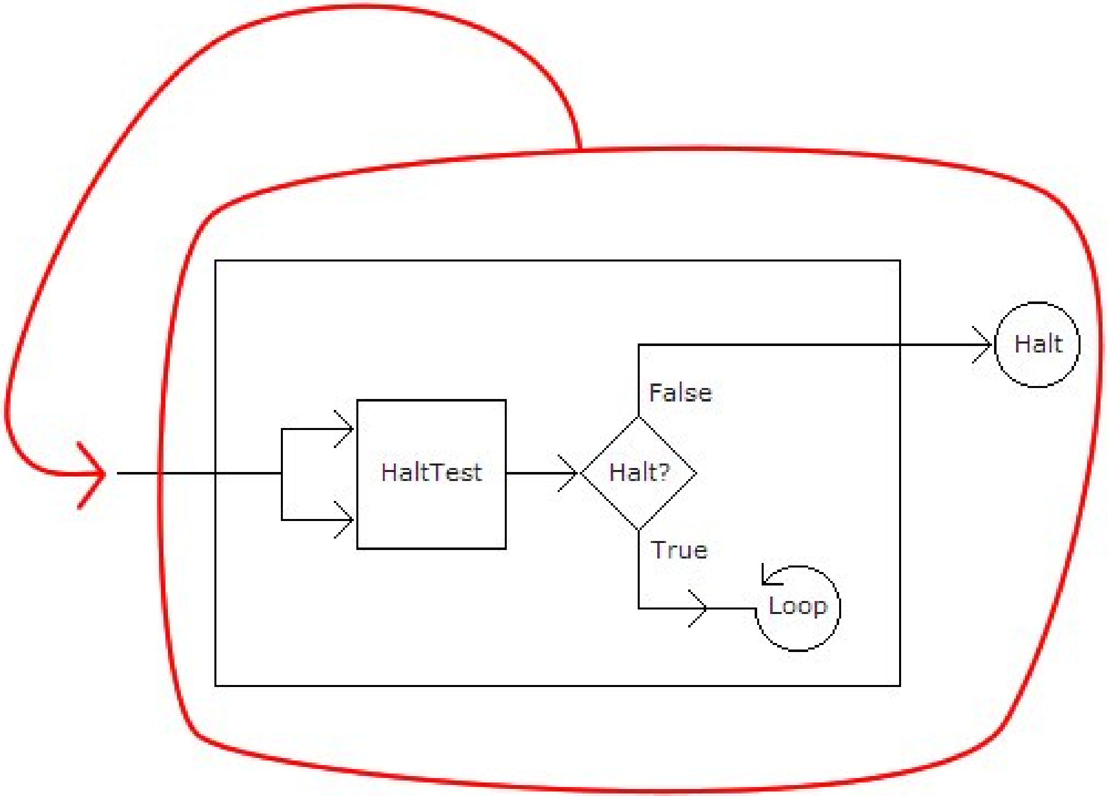
Undecidable Queries

- The deep reason behind the undecidability of first order logic is the fact that its expressive power is so great that it is possible to state undecidable queries.
- One of the famous undecidable queries is the halting problem.
- Here is what a halting algorithm would look like:
  - Input: a specification of a computational procedure P, and an input I for that procedure
  - Output: an answer 'halt' if P halts when applied to I, and 'loop' otherwise.

Undecidability of the Halting Problem, in pictures . . .



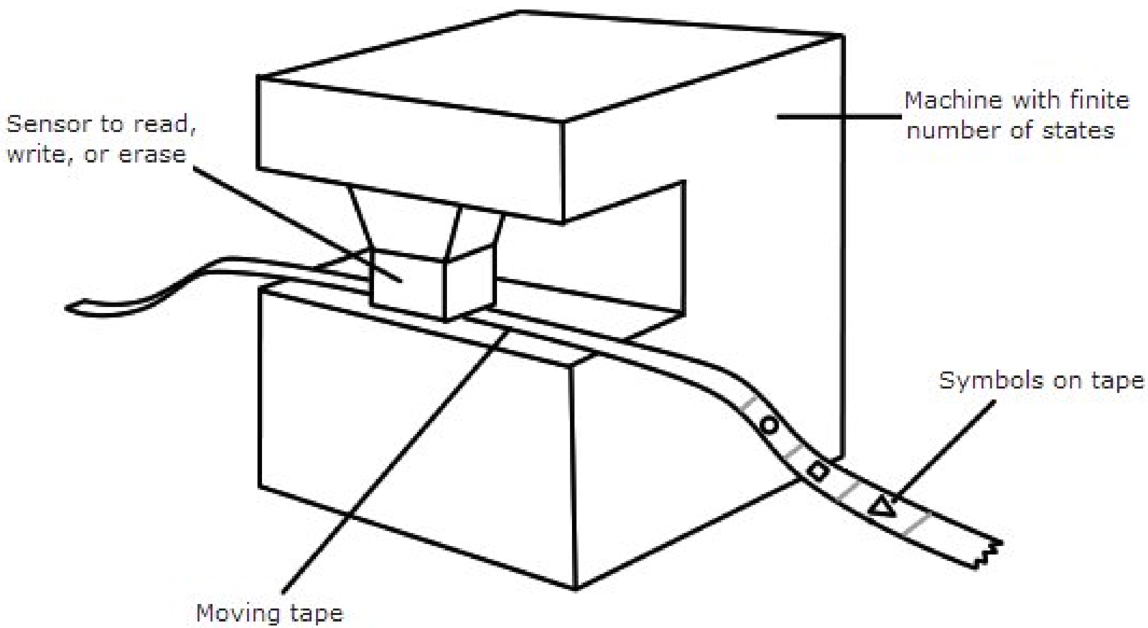



HaltTest
Halt?
False
True
Halt
Loop

Alan Turing's Insight

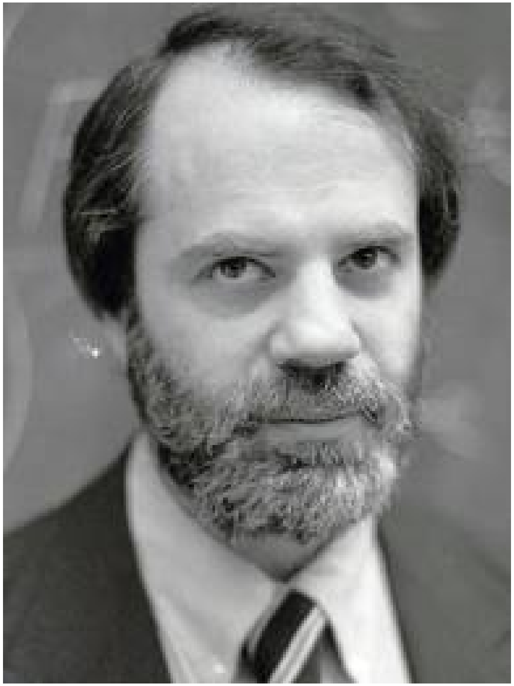
- A language that allows the specification of 'universal procedures' such as HaltTest cannot be decidable.
- But first order predicate logic is such a language . . .
- The formal proof of the undecidability of first order logic consists of
  - A **very general** definition of computational procedures.
  - A demonstration of the fact that such computational procedures can be expressed in first order logic.
  - A demonstration of the fact that the halting problem for computational procedures is undecidable (see the picture above).
  - A formulation of the halting problem in first order logic.
- This formal proof was provided by Alan Turing in [**?**]. The computational procedures he defined for this purpose were later called Turing machines.

Picture of a Turing Machine

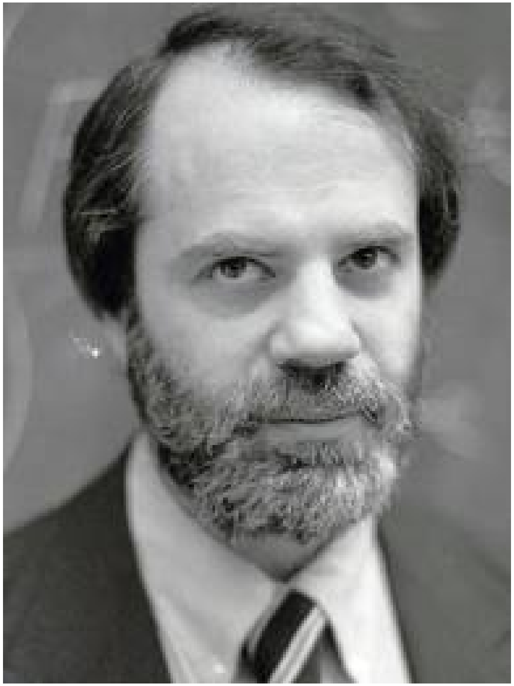


A Turing Machine

Who is Who in Modal and Epistemic Logic?



Who is Who in Modal and Epistemic Logic?



Saul Kripke (born 1940) Jaakko Hintikka (born 1929)

Epistemic Logic

Equivalence relations as partitions:

A partition β of a set X is a family of subsets of X with the following properties:

1. $\bigcup \beta = X$,
2. $Y \in \beta$ implies $Y \neq \emptyset$,
3. $Y, Z \in \beta \wedge Y \neq Z$ implies $Y \cap Z = \emptyset$.

Partitions of X correspond to equivalence relations on X in the following precise sense:

- If $\sim$ is an equivalence on X, then $\{[x]_\sim \mid x \in X\}$ is the corresponding partition.
- If β is a partition of X, then $\sim_\beta$ given by $x \sim_\beta y$ iff $\exists Y \in \beta$ such that $\{x, y\} \subseteq Y$ is the corresponding equivalence relation on X.

Building epistemic models from partitions

```
type Erel a = [[a]]
```

The block of an element in a partition:

```
bl :: Eq a => Erel a -> a -> [a]
bl r x = head (filter (elem x) r)
```

The restriction of a partition to a domain:

```
restrict :: Ord a => [a] -> Erel a -> Erel a
restrict domain =  nub . filter (/= [])
                   . map (filter (flip elem domain))
```

Agents and Basic Propositions

```
data Agent = Ag Int deriving (Eq,Ord)

a,b,c,d,e :: Agent
a = Ag 0; b = Ag 1; c = Ag 2; d = Ag 3; e = Ag 4
```

```
data Prp = P Int | Q Int | R Int | S Int
  deriving (Eq,Ord)
instance Show Prp where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
  show (S 0) = "s"; show (S i) = "s" ++ show i
```

Epistemic models

```
data EpistM state = Mo
                [state]
                [Agent]
                [(state,[Prp])]
                [(Agent,Erel state)]
                [state]  deriving (Eq,Show)
```

```
example1 :: EpistM Int
example1 = Mo
  [0..3]
  [a,b,c]
  []
  [(a,[[0],[1],[2],[3]]),
   (b,[[0],[1],[2],[3]]),(c,[[0..3]])]
  [1]
```

```
example2 :: EpistM Int
example2 = Mo
  [0..3]
  [a,b,c]
  [(0,[P 0,Q 0]),(1,[P 0]),(2,[Q 0]),(3,[])]
  [(a,[[0..3]]),(b,[[0..3]]),(c,[[0..3]])]
  [0..3]
```

Extracting an epistemic relation from a model

```
rel :: Agent -> EpistM a -> Erel a
rel ag (Mo _ _ _ rels _) = apply rels ag
```

Epistemic Formulas

```
data Frm a = Tp
           | Info a
           | Prp Prp
           | N (Frm a)
           | C [Frm a]
           | D [Frm a]
           | Kn Agent (Frm a)
          deriving (Eq,Ord,Show)
```

Truth Definition

...

```
isTrueAt :: Ord state =>
            EpistM state -> state -> Frm state -> Bool
isTrueAt m w Tp = True
isTrueAt m w (Info x) = w == x
isTrueAt
  m@(Mo worlds agents val acc points) w (Prp p) = let
   props = apply val w
  in
   elem p props
isTrueAt m w (N f) = not (isTrueAt m w f)
isTrueAt m w (C fs) = and (map (isTrueAt m w) fs)
isTrueAt m w (D fs) = or  (map (isTrueAt m w) fs)
isTrueAt
 m@(Mo worlds agents val acc points) w (Kn ag f) = let
    r = rel ag m
    b = bl r w
  in
    and (map (flip (isTrueAt m) f) b)
```

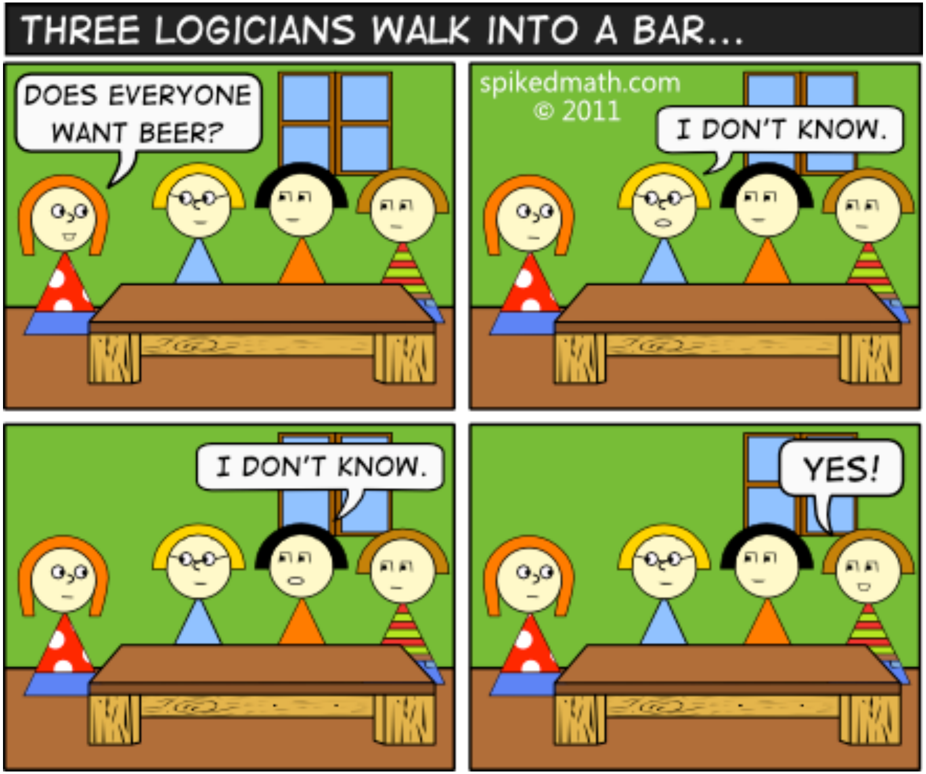
Public Announcement

Restriction to φ worlds:

```
upd_pa :: Ord state =>
          EpistM state -> Frm state -> EpistM state
upd_pa m@(Mo states agents val rels actual) f =
   (Mo sts' agents val' rels' actual')
    where
    sts'    = [ s | s <- states, isTrueAt m s f ]
    val'    = [ (s, ps) | (s,ps) <- val, s `elem` sts']
    rels'   = [(ag,restrict sts' r) | (ag,r) <- rels ]
    actual'= [ s | s <- actual, s `elem` sts' ]
```

```
upds_pa ::  Ord state =>
          EpistM state -> [Frm state] -> EpistM state
upds_pa = foldl upd_pa
```

Three Logicians



```
bools = [True,False]
```

Initialize the bar situation: they all know what they want but are ignorant about what the others want.

```
initBar :: EpistM (Bool,Bool,Bool)
initBar = Mo states [a,b,c] [] rels [(True,True,True)]
  where
  states = [ (b1,b2,b3) | b1 <- bools,
                          b2 <- bools,
                          b3 <- bools ]
  rela = (a,[[(True,x,y)  | x <- bools, y <- bools],
             [(False,x,y) | x <- bools, y <- bools]])
  relb = (b,[[(x,True,y)  | x <- bools, y <- bools],
             [(x,False,y) | x <- bools, y <- bools]])
  relc = (c,[[(x,y,True)  | x <- bools, y <- bools],
             [(x,y,False) | x <- bools, y <- bools]])
  rels = [rela,relb,relc]
```

Statements of ignorance and knowledge:

```
allBeer :: Frm (Bool,Bool,Bool)
allBeer = Info (True,True,True)

ignA, ignB, ignC :: Frm  (Bool,Bool,Bool)
ignA = C [N (Kn a allBeer), N (Kn a (N allBeer))]
ignB = C [N (Kn b allBeer), N (Kn b (N allBeer))]
ignC = C [N (Kn c allBeer), N (Kn c (N allBeer))]

knowC, knowC' :: Frm  (Bool,Bool,Bool)
knowC  = Kn c allBeer
knowC' = Kn c (N allBeer)
```

Finally, Updating

```
barModel1 = upd_pa initBar ignA
```

Result of second update:

```
barModel2 = upd_pa barModel1 ignB
```

Result of third update:

```
barModel3 = upd_pa barModel2 knowC
```

Or the third logician could have said ‘no’, for ‘I know that not all of us want beer’.

```
barModel3' = upd_pa barModel2 knowC'
```

Links and Books for Further Study

http://www.logicinaction.org

http://www.cwi.nl/~jve/HR

http://projecteuler.net

http://www.cwi.nl/~jve/software/demo_s5