Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate

Corso di laurea triennale in Informatica

Implementazione di un calcolo a Tableaux per la logica proposizionale classica con integrazione di property base testing

Tesi di Laurea di:

Francesco Mancuso

Matr. n. 729183

Relatore: Prof. **Mauro Ferrari**

Anno Accademico 2018 – 2019

A tutte le persone che hanno creduto in me.

Sommario

Riassunto

In questo lavoro di tesi ho realizzato un applicazione, tramite il linguaggio di programmazione funzionale orientato agli oggetti Scala, per l'implementazione dei Tableaux per la logica proposizionale classica con integrazione di property base testing. Nella prima parte del lavoro vengono fornite diverse definizioni al fine di fornire le basi per comprendere i Tableaux.

Vengono fornite le nozioni base della logica proposizionale (valutazione delle variabili, formule proposizionali e formule segnate) al fine di introdurre il concetto di Tableaux.

Successivamente viene presentata la definizione di Tableaux e descritta la sua costruzione formale illustrandone alcuni esempi del suo uso.
Viene introdotto il framework di property based testing ScalaCheck e le nozioni di proprietà e di generatore, affiancate da degli esempi d'uso.

Nella seconda parte, viene illustrata l'implementazione in Scala dell'applicazione e viene fornita una spiegazione per ogni sua componente.
Viene mostrata l'implementazione delle formule proposizionali, delle formule segnate e della struttura del Tableaux accompagnata dal suo algoritmo di costruzione.
Viene illustrata l'implementazione delle proprietà di ScalaCheck definite sugli alberi binari.

L'applicazione realizzata prende in input un file di testo al cui interno viene definita una formula proposizionale. Dal file viene estratta la formula per essere in seguito scansionata e rappresentata come albero binario di oggetti. L'albero binario viene scorso per costruire ricorsivamente il suo Tableaux, che a sua volta è un albero binario. Alla fine del processo vengono controllate le foglie per verificare se la formula proposizionale di partenza è verificata (tautologia).

Introduzione

In questo lavoro di tesi ho realizzato un applicazione, tramite il linguaggio di programmazione funzionale orientato agli oggetti Scala, per l'implementazione dei Tableaux.

La programmazione funzionale si basa sul concetto di funzione e prevede che possano essere passate come argomenti ad altre funzioni e restituite come valore di una funzione.

Scala prende molte delle sue caratteristiche funzionali da linguaggi come Haskell e Erlang. Inoltre è legato a Java, infatti la compilazione del codice Scala produce Bytecode che viene dato in esecuzione ad una Java Virtual Machine.

In Scala non c'è una distinzione tra tipi primitivi (int, char, boolean, byte, short, double, float) e tipi referenziati (classi e interfacce), infatti ogni cosa (numeri, caratteri,…) è un oggetto.

Una buona programmazione in Scala prevede l'utilizzo di valori immutabili, ovvero valori che non possono essere modificati. Tuttavia, è prevista la creazione di variabili mediante la parola chiave "var".

Nel primo capitolo del lavoro di tesi è presentata la logica proposizionale attraverso definizioni introduttive (valutazione delle variabili e formule) volte a fornire le basi per la comprensione, nel capitolo successivo, dei Tableaux e dello loro applicazione. Nel secondo capitolo viene definito il concetto di formula segnata, di Tableaux e viene illustrata formalmente la sua costruzione applicando delle specifiche regole (α-formule e β–formule).

Nel terzo capitolo viene descritto il framework di property based testing ScalaCheck. Viene fornita la nozione di proprietà, di generatore e vengono illustrati degli esempi d'uso.

Dal quarto capitolo in poi, viene illustrata l'implementazione vera e propria delle formule proposizionali, dei Tableaux e delle proprietà di ScalaCheck, accompagnata da un commento descrittivo.

L'applicazione realizzata prende in input un file di testo al cui interno viene definita una formula proposizionale. Dal file viene estratta la formula per essere in seguito scansionata e rappresentata come albero binario di oggetti.

L'albero binario viene scorso per costruire ricorsivamente il suo Tableaux, che a sua volta è un albero binario. Alla fine del processo vengono controllate le foglie per verificare se la formula proposizionale di partenza è verificata (tautologia).

Alla fine viene utilizzato ScalaCheck per definire delle proprietà sugli alberi binari che rappresentano le formule proposizionali.

Formule della logica proposizionale

La logica proposizionale è quella branca della matematica che si occupa dello studio del valore di verità di affermazioni. E' un linguaggio formale basato su proposizioni elementari (variabili) e operatori logici (And, Or, Implicazione e Negazione), che restituiscono il valore di verità in base al valore di verità associato alle componenti che ne fanno parte.
La struttura delle proposizioni si basa su un alfabeto così composto:

1) Simboli proposizionali (variabili)
2) Connettivi logici
3) Parentesi

Variabili proposizionali

Definizione Le variabili proposizionali sono un insieme numerabile di simboli P1, P2, …, Pn.

Tavole di verità

Prima di definire cos'è una tavola di verità, bisogna intrudurre il concetto di valutazione di una formula proposizionale.

Definizione Una valutazione è una funzione v: formula → {0,1} che ad una formula proposizionale associa un valore di verità true o false.

La sua definizione è definita come segue:

1) Se P è variabile proposizionale , allora v(P) = v(P)
2) Se P è ¬X, allora v(¬X) = 1 – v(X)
3) Se P è (X1 & X2), allora v(X1 & X2) = min(v(X1), v(X2))
4) Se P è (X1 || X2), allora v(X2 || X2) = max(v(X1), v(X2))
5) Se P è (X1 => X2), allora v(X2 => X1) = max(1 – v(X1), v(X2))

Definizione Le tavole di verità sono delle tabelle in cui, assegnati dei valori di verità alle componenti proposizionali che le compongono, viene determinata se una formula proposizionale è vera o falsa.

Operatori logici

I principali operatori logici utilizzati nelle formule proposizionali sono i seguenti: & (congiunzione), || (disgiunzione), => (implicazione) e ¬(negazione).

Le loro tavole di verità sono le seguenti:

		&
False	False	False
False	True	False
True	False	False
True	True	True

False	False	False
False	True	True
True	False	True
True	True	True

		=>
False	False	True
False	True	True
True	False	False
True	True	True

	¬
False	True
True	False

Parentesi

I simboli "(" e ")" vengono chiamati rispettivamente parentesi sinistra e parentesi destra. Il loro scopo è quello di rendere il linguaggio più chiaro evitando ambiguità.

Formule

Definizione La nozione di formula è definita dalle seguenti regole ricorsive:

1. Ogni variabile proposizionale è una formula
2. Se A è una formula, anche ¬A lo è
3. Se A, B sono formule, lo sono anche le combinazioni di queste formule con i connettivi logici: (A & B), (A || B), (A => B)

Esempio di formula proposizionale: ((A & B) => (C || B)) || (¬D)

Come detto precedentemente, le parentesi vengono utilizzate per rendere chiaro il linguaggio. Tuttavia, in mancanza di queste bisogna definire delle regole di precedenza degli operatori logici. Stabiliamo le precedenze nei seguenti modi:

Operatore	Precedenza
¬	1
&	2
||	3
=>	4

La seguente formula senza parentesi: A & B => ¬C || D

con le regole di precedenza definite precedentemente si legge in questo modo:

(A & B) => (¬C || D)

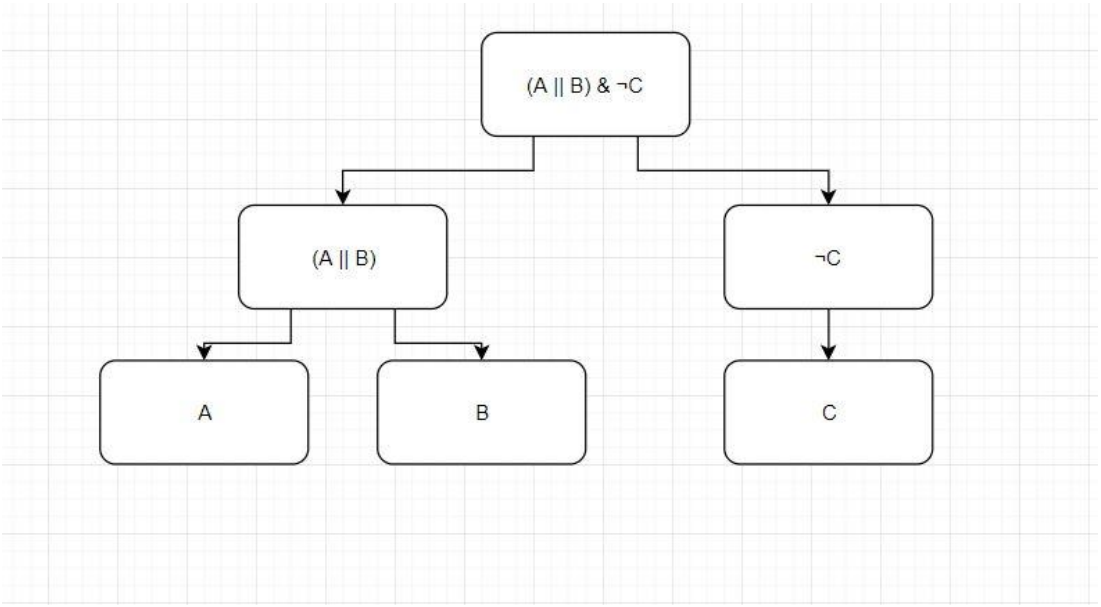
Definizione Le sotto-formule di una formula A sono definite nel seguente modo:

1. Se A è una variabile proposizionale, allora A è sotto-formula di A
2. Se A = ¬A1, allora A1 è sotto-formula di A
3. Se A = A1 # A2 (con # connettivo logico binario), allora A1 e A2 sono sotto-formule di A

Una formula A può essere descritta da un albero i cui nodi sono etichettati da formule. L'albero è così costruito:

1. La radice dell'albero è la formula A.
2. Se un nodo è etichettato dalla formula B con connettivo logico principale binario, allora i suoi due nodi figli sono formati dalle due sotto-formule di B.
3. Se un nodo è etichettato dalla formula B con connettivo logico unario principale, allora il suo nodo figlio è etichettato dalla sotto-formula principale di B.
4. Le foglie sono etichettate da variabili proposizionali.

Esempio: X = (A || B) & ¬C



Definizione Una formula X si dice soddisfacibile se esiste una valutazione per le sue variabili tale che v(X) = true

Esempio: X = (A & B) || B

A	B	A & B	(A & B) || B
False	False	False	False
False	True	False	True
True	False	False	False
True	True	True	True

In questo esempio, la formula X è soddisfacibile perché v(X) = 1 per la seconda e quarta riga della tabella di verità.

Definizione Una formula X è una tautologia se per ogni valutazione delle sue variabili si ha v(X) = true
Esempio: X = A || ¬A

A	¬A	A || ¬A
False	True	True
True	False	True
False	True	True
False	False	True

In questo esempio, la formula X è una tautologia perché v(X) = true per ogni riga della tabella di verità.

Definizione Una formula X è insoddisfacibile se per ogni valutazione delle sue variabili si ha v(X) = false

Esempio: X = A & ¬A

A	¬A	A & ¬A
False	True	False
True	False	False
False	True	False
True	False	False

In questo esempio, la formula X è insoddisfacibile perché v(X) = false per ogni riga della tabella di verità.

Tableaux

Per verificare se una formula con n variabili è una tautologia bisogna costruire una tavola di verità con 2^n righe. Per formule con elevato numero di variabili diventa computazionalmente svantaggioso (complessità esponenziale) controllare se sono tautologie.

Un altro metodo per verificare se una formula è una tautologia è il metodo dei Tableaux. Alla base dei Tableaux ci sono i seguenti quattro fatti:

1. ¬A è vero se e solo se A è falso.
2. A & B è vero se e solo se A e B sono veri.
3. A || B è vero se e solo se almeno uno tra A e B è vero.
4. A => B è falso se e solo se A è falso e B è vero.

Questi fatti possono essere rappresentati come regole mediante l'introduzione di formule segnate.

Definizione Una formula segnata si compone di un simbolo (T o F) e una formula X. Una formula segnata TX è vera se e solo se X è vera.

Una formula segnata FX è vera se e solo se X è falsa.

Definizione Una coppia di letterali TX, FX si dice complementare**.**

Definzione Un Tableaux per una formula X, è un albero G i cui ogni nodo è etichettato da un insieme di sotto-formule di X. L'albero si costruisce per passi successivi applicando le regole in tabella a partire dalla radice etichettata {FX}.

Le regole per la costruzione dei Tableaux si dividono in due tipi: quelle che realizzano una biforcazione (β - formule) e quelle che hanno una diretta conseguenza (α – formule):

	Regole per la costruzione dei Tableaux	
		Ridotti

1	S, F(X & Y)	S, FX | S, FY
2	S, T(X || Y)	S, TX | S, TY
3	S, T(X => Y)	S, FX | S, TY
4	S, T(X & Y)	S, TX, TY
5	S, F(X || Y)	S, FX, FY
6	S, F(X => Y)	S, TX, FY
7	S, T¬X	S, FX
8	S, F¬X	S, TX

Nota: Il simbolo "|" indica che nel passo di riduzione vengono costruiti due nodi figli con ciascuno uno dei due ridotti. Il simbolo "," indica che nel passo di riduzione viene costruito un nodo figlio con entrambi i ridotti.

La procedura di costruzione dei Tableaux è la seguente:

Passo 1

1. Costruiamo l'albero G(i) dalla sola radice etichettata con {FX}

Passo i (i > 1)

Se al passo i - 1 abbiamo costruito l'albero G(i - 1), al passo i costruiamo l'albero G(i) agendo sulle foglie dell'albero G(i – 1):

1. Se nelle foglie ci sono solo variabili proposizionali con segno (letterali), allora la costruzione termina e G(i – 1) è l'albero finale.
2. Se invece in una foglia è presente una formula segnata H che non è un letterale, allora:
    a. Se X = F¬X(1), allora l'albero G(i) si costruisce aggiungendo un nodo figlio "n" con etichetta ("E" rappresenta l'etichetta del nodo)
    E(n) = (E(G(i – 1)) \ {X}) U {TX(1)}
    b. Se X è un α – formula con ridotti X(1) e X(2), allora il sotto-albero G(i) si costruisce aggiungendo un nodo figlio "n" all'albero G(i – 1) con etichetta
    E(n) = (E(G(i – 1)) \ {X}) U {X(1), X(2)}
    c. Se X è una β – formula con ridotti X(1) e X(2), allora il sotto-albero G(i) si costruisce aggiungendo due nodi figli "n1" e "n2" all'albero G(i – 1) con etichetta
    E(n1) = (E(G(i – 1)) \ {X}) U {X(1)}
    E(n2) = (E(G(i – 1)) \ {X}) U {X(2)}

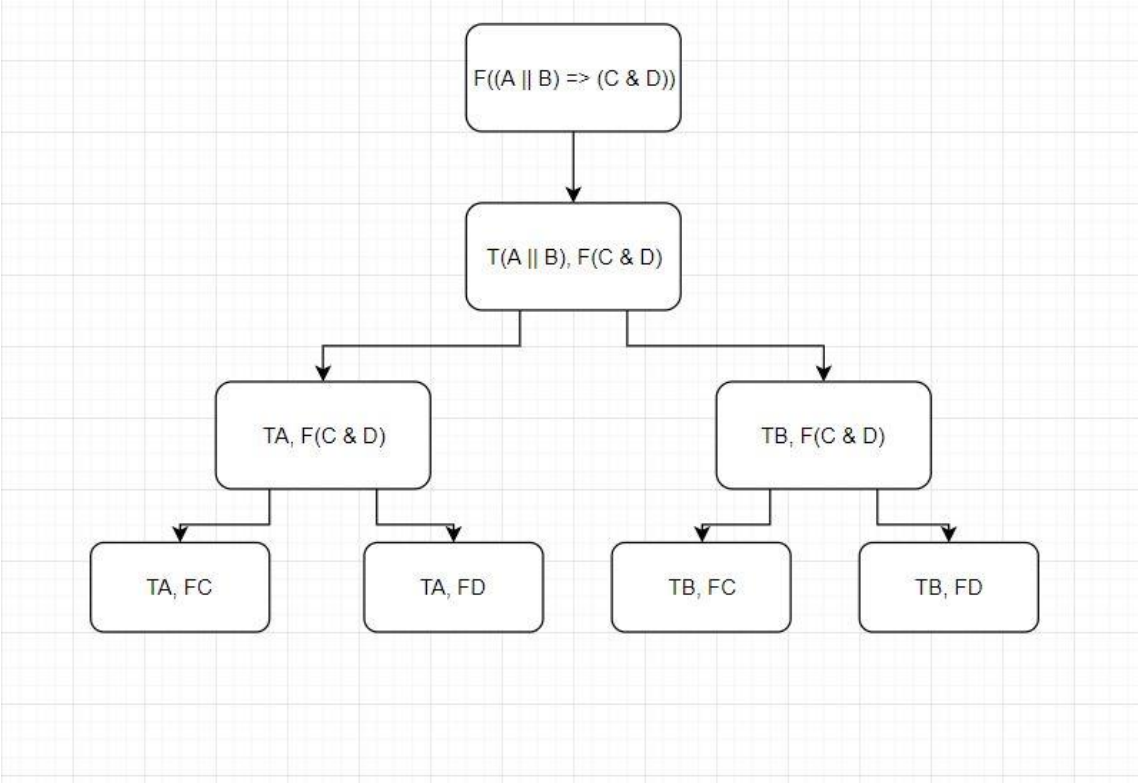
Definizione Un ramo di un Tableaux si dice chiuso se la sua foglia contiene almeno una coppia complementare. Un Tableaux si dice chiuso se tutti i rami sono chiusi.

Definzione Una formula X è verificata (è una tautologia) se e solo se il Tableaux di FX è chiuso.

Per esempio:

Supponiamo di voler dimostrare la formula ((A || B) => (C & D)). Per dimostrarla supponiamo che la formula sia falsa (anteponendo il simbolo F).

Di seguito è rappresentato il Tableaux della formula sopracitata:



La sua costruzione avviene mediante le seguenti fasi:

1) Questa è la radice dell'albero:

   F((A || B) => (C & D))

2) In questo passo ho applicato la regola 6 (α – formula):

   T(A || B), F(C & D)

3) In questo passo ho applicato la regola 2 (β – formula):

   TA, F(C & D) | TB, F(C & D)

4) In questo passo ho applicato la regola 1 (β – formula) semplificando in un passaggio la biforcazione dei due nodi del passo precedente:

TA, FC | TA, FD TB, FC | TB, FD

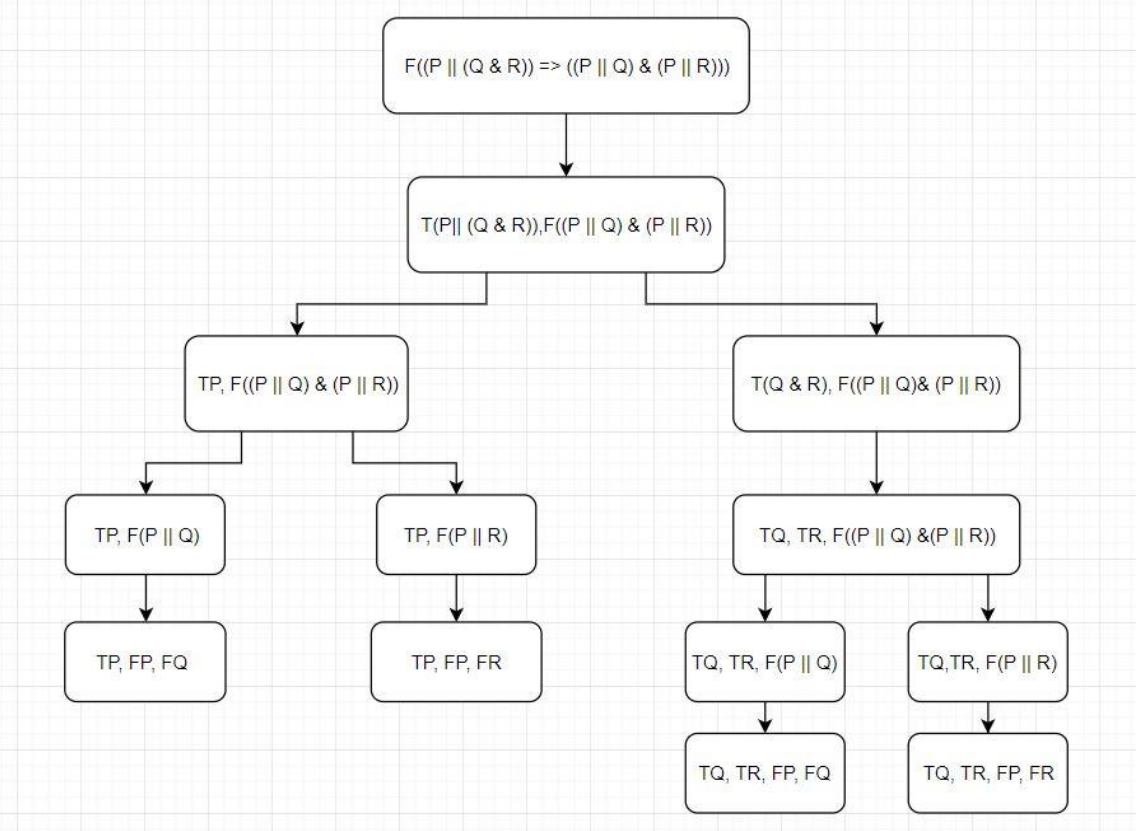
5) La costruzione termina perché nei nodi ottenuti precedentemente sono presenti solo letterali.
6) Nelle quattro foglie non sono presenti coppie complementari quindi la formula iniziale è insoddisfacibile. Se in tutte le foglie fossero presenti delle coppie complementari, allora in ogni foglia ci sarebbe una contraddizione. Questo significa che la formula è sempre vera per ogni valutazione, ovvero è una tautologia.

Un altro esempio:

Supponiamo di voler dimostrare la formula (P || (Q & R)) => ((P || Q) & (P || R**)).**

Per dimostrarla supponiamo che la formula sia falsa (anteponendo il simbolo F).

Di seguito è rappresentato il Tableaux della formula sopracitata:



La sua costruzione avviene mediante le seguenti fasi:

1) Questa è la radice dell'albero:

   F((P || (Q & R)) => ((P || Q) & (P || R)))

2) In questo passo ho applicato la regola 5 (α – formula):

T(P || (Q & R)), F((P || Q) & (P || R))

3) In questo passo ho applicato la regola 2 (β – formula):

   TP, F((P || Q) & (P || R)) | T(Q & R), F((P || Q) & (P || R))

4) In questo passo ho applicato la regola 1 (β – formula):

   TP, F(P || Q) | TP, F(P || R) | T(Q & R), F((P || Q) & (P || R))

5) In questo passo ho applicato la regola 5 (α – formula):

   TP, FP, FQ | TP, F(P || R) | T(Q & R), F((P || Q) & (P || R))

6) In questo passo ho applicato la regola 5 (α formula):

   TP, FP, FQ | TP, FP, FR | T(Q & R), F((P || Q) & (P || R))

7) In questo passo ho applicato la regola 4 (α – formula):

   TP, FP, FQ | TP, FP, FR | TQ, TR, F((P || Q) & (P || R))

8) In questo passo ho applicato la regola 1 (β – formula):

   TP, FP, FQ | TP, FP, FR TQ, TR, F(P || Q) | TQ, TR, F(P || R)

9) In questo passo ho applicato la regola 5 (α – formula):

   TP, FP, FQ | TP, FP, FR TQ, TR, FP, FQ | TQ, TR, F(P || R)

10) In questo passo ho applicato la regola 5 (α – formula):

    TP, FP, FQ | TP, FP, FR TQ, TR, FP, FQ | TQ, TR, FP, FR

11) La costruzione termina perché nei nodi ottenuti precedentemente sono presenti solo letterali.
12) Nelle quattro foglie sono presenti coppie complementari (TP – FP, TP – FP, TQ – FQ, TR – FR) quindi la formula iniziale è una tautologia.

ScalaCheck

ScalaCheck è una libreria scritta in Scala usata per la generazione di test automatici basati su proprietà. Una proprietà è un unità testabile identificata dalla classe org.scalacheck.Prop.
ScalaCheck, data una proprietà che descrive il comportamento atteso, genererà dati di input casuali, coprendo molti casi limite e controllando il risultato.
Esempio di proprietà: size(t) == size(t1) + size(t2)

Il numero di nodi dell'albero t è uguale al numero di nodi del sotto-albero t1 più il numero di nodi del sotto-albero t2. Questa è una proprietà che vale per tutti gli alberi binari.

Per creare proprietà viene usato il metodo org.scalacheck.forAll.
Il metodo forAll crea proprietà universalmente quantificate, prende una funzione come parametro e crea una proprietà che può essere testata con il metodo check. ForAll ritorna true o false e può prendere parametri di diverso tipo come interi, stringhe, liste, ecc… su cui viene testata la proprietà.

Per esempio:

```
val propConcatList = forAll{ (l: List[Int], l2: List[Int]) =>
        (l.size + l2.size) == (l ::: l2).size
}
propConcatList.check
```

Questa proprietà verifica che, prese due liste qualsiasi, la somma della loro lunghezza è uguale alla lunghezza della lista ottenuta dalla loro concatenazione. Chiaramente ScalaCheck restituirà true perché questa proprietà è verificata per tutte le possibili liste di numeri interi.

Un altro esempio:

```
val propSqrt = forAll{ (n: Int) =>
               Math.sqrt(n*n) == n*n
}
propSqrt.check
```

Questa proprietà verifica che per ogni numero intero n , la radice quadrata di n*n è uguale a n*n. In questo caso ScalaCheck restituirà false, perché esistono dei numeri interi che falsificano la proprietà. Come n = -1.

E' possibile raggruppare diverse proprietà in un object estendendo la classe Properties.
Esempio:

```
import Prop.forAll

object StringSpecification extends Properties("String") {

  property("startsWith") = forAll { (a: String, b: String) =>
    (a+b).startsWith(a)
  }
```

```
  property("endsWith") = forAll { (a: String, b: String) =>
    (a+b).endsWith(b)
  }
}
```

StringSpecification è un oggetto che contiene due proprietà che caratterizzano le stringhe. La prima proprietà dice che, date due stringhe "a" e "b", la concatenazione di loro inizia con la stringa "a".
La seconda proprietà dice che, date due stringhe "a" e "b", la concatenazione di loro finisce con la stringa "b".
Per verificarle bisogna dichiarare una variabile

```
val checkProp: Seq[(String, Test.Result)] =
Test.checkProperties(Test.Parameters.default, this)
```

e testarla ciclando su tutte le proprietà dichiarate e controllandone l'esito:

```
val tests = StringSpecification
tests.checkProp.foreach(el => {
   println("(" + el._1 + ")" + " => " + el._2.passed + " : " +
el._2.status)
 })
```

Il risultato è il seguente:

```
(String.startsWith) => true : Passed
(String.endsWith) => true : Passed
```

Per personalizzare le proprietà, si possono introdurre i generatori.
I generatori sono responsabili della generazione dei dati di test in ScalaCheck e sono rappresentati dalla classe org.scalacheck.Gen. È necessario sapere come utilizzare questa classe se si desidera che ScalaCheck generi dati di tipi che non sono di base supportati. Nell'oggetto Gen esistono diversi metodi per creare nuovi e modificare generatori esistenti.

Per esempio:

```
val gen = Gen.choose(1, 100)
val propSqrt = forAll(gen){(n: Int) =>
        Math.sqrt(n*n) == n*n
}
propSqrt.check
```

In questo esempio ho definito un semplice generatore che genera dei numeri compresi tra 1 e 100. Utilizzando questo generatore nella proprietà sulle radici quadrate descritta precedentemente, ScalaCheck restituirà true perché nei test che effettua verranno esclusi tutti i numeri negativi che invalidavano la proprietà.

E' possibile realizzare generatori più complicati che permettono, per esempio, la creazione di alberi binari casuali (generando case class).

Per esempio:

```
sealed abstract class Tree
case class Node(left: Tree, right: Tree, v: Int) extends Tree
case object Leaf extends Tree

import org.scalacheck._
import Gen._
import Arbitrary.arbitrary

val genLeaf = const(Leaf)

val genNode = for {
  v <- arbitrary[Int]
  left <- genTree
  right <- genTree
} yield Node(left, right, v)

def genTree: Gen[Tree] = oneOf(genLeaf, genNode)
```

Il metodo oneOf crea un generatore che casualmente prende uno dei suoi parametri ogni volta che genera un valore.

Un possibile albero binario generato dalla costruzione di case class annidate può avere questa forma:

Node(Leaf, Node(Leaf, Node(Leaf, Leaf, 100), -10), 50)

Sempre in questo esempio viene introdotto un generatore speciale chiamato arbitrary. Questo generatore genera valori arbitrari di un tipo supportato.

Implementazione

FormulaWff

```
import scala.annotation.tailrec


sealed trait FormulaWff {


 override def toString: String = this match {
  case Not(e1) => "¬(" + e1 + ")"
  case And(e1, e2) => "(" + e1 + " & " + e2 + ")"
  case Or(e1, e2) => "(" + e1 + " | " + e2 + ")"
  case Implies(e1, e2) => "(" + e1 + " -> " + e2 + ")"
  case Var(e1) => e1
 }

 def size: Int = {

  @tailrec
  def iter(l: List[FormulaWff], count: Int): Int =
   l match {
    case Nil => count
    case Var(_) :: ls => iter(ls, count + 1)
    case Not(e1) :: ls => iter(e1 :: ls, count + 1)
    case And(e1, e2) :: ls => iter(e1 :: e2 :: ls, count + 1)
    case Or(e1, e2) :: ls => iter(e1 :: e2 :: ls, count + 1)
    case Implies(e1, e2) :: ls => iter(e1 :: e2:: ls, count + 1)

   }
  iter(List(this), 0)
 }

 def depth: Int = {

  def iter(f: FormulaWff): Int = f match {
   case Var(_) => 0
   case Not(e1) => 1 + iter(e1)
   case And(e1, e2)  => 1 + Math.max(iter(e1), iter(e2))
   case Or(e1, e2) => 1 + Math.max(iter(e1), iter(e2))
   case Implies(e1, e2) => 1 + Math.max(iter(e1), iter(e2))
  }

  iter(this)
 }

 def leaves: Int = {
```

```
    @tailrec
    def iter(l: List[FormulaWff], count: Int): Int =
      l match {
        case Nil => count
        case Var(_) :: ls => iter(ls, count + 1)
        case Not(e1) :: ls => iter(e1 :: ls, count)
        case And(e1, e2) :: ls => iter(e1 :: e2 :: ls, count)
        case Or(e1, e2) :: ls => iter(e1 :: e2 :: ls, count)
        case Implies(e1, e2) :: ls => iter(e1 :: e2 :: ls, count)

      }
    iter(List(this), 0)
  }

}



case class And(f1: FormulaWff, f2: FormulaWff) extends FormulaWff

case class Or(f1: FormulaWff, f2: FormulaWff) extends FormulaWff

case class Implies(f1: FormulaWff, f2: FormulaWff) extends FormulaWff

case class Not(f1: FormulaWff) extends FormulaWff

case class Var(s: String) extends FormulaWff
```

Le formule proposizionali sono state rappresentate come alberi binari definendo un trait FormulaWff e delle case class identificanti gli operatori logici che lo estendono. Ci sono dei metodi comuni a tutti gli operatori logici:

1) **Leaves** => restituisce il numero di foglie dell'albero rappresentante la formula proposizionale.
2) **Size** => restituisce il numero di nodi dell'albero rappresentante la formula proposizionale.
3) **Depth** => restituisce il cammino più lungo dalla radice (depth = 0) alle foglie.
4) **ToString** => restituisce la rappresentazione grafica della formula proposizionale.

I metodi **Leaves** e **Size** sono tail – recursive. Sono metodi ricorsivi in cui la chiamata a se stessa avviene come ultima istruzione. Questo permette il riutilizzo di record di attivazione nello stack, così da evitare l'errore di StackOverflow per ricorsioni molto profonde.

Due esempi di costruzione di una formula proposizionale:

1) ***Val f1 = And(Or(Var("A"), Var("C")), Var("D"))***
   Es: f1 = ((A || C) & D)
2) ***Val f2 = Implies(Not(Var("A")), And(Var("C"), Var("D"))***
   Es: f2 = (¬A => (C & D))

SignedFormula

```
object EnumFormula extends Enumeration {
  val Alpha, Beta, Var = Value
}

sealed trait Sign{

  override def toString: String = this match{
    case T => "T"
    case F => "F"
  }
}



case object T extends Sign

case object F extends Sign


sealed trait SignedFormula{

  def getType: EnumFormula.Value = this match{
    case SF(T, And(_,_)) |
         SF(F, Or(_,_)) |
         SF(F, Implies(_,_)) |
         SF(F, Not(_)) |
         SF(T, Not(_)) => EnumFormula.Alpha
    case SF(T, Or(_,_)) |
         SF(F, And(_,_)) |
         SF(T, Implies(_,_)) => EnumFormula.Beta
    case SF(_,Var(_)) => EnumFormula.Var
  }

  override def toString: String = this match {
    case SF(T, f) => T + "(" + f + ")"
    case SF(F, f) => F + "(" + f + ")"
  }

}

case class SF(s: Sign, f: FormulaWff) extends SignedFormula
```

Le formule segnate sono state realizzate definendo un trait SignedFormula e una case class SF che lo estende. La classe SF (quella che rappresenta la formula segnata) è costruita con una formula proposizionale e il suo segno.
I segni sono identificati da due object: T e F.
All'interno del trait è stato definito un metodo **getType** che restituisce il tipo di formula segnata (α, β o variabile proposizionale).

Due esempi di costruzione di una formula segnata:

1) ***Val s1 = SF(T, And(Var("A"), Var("B"))*** *Es: s1 = T(A & B)*
2) ***Val s2 = SF(F, Or(Var("C"), Var("D"))*** *Es: s2 = F(C || D)*

Tableaux

```
sealed trait Tableaux{

 def isClosed: Boolean = {

  def iter(t: Tableaux): Boolean = t match{
   case AlphaNode(_,s,_) => iter(s)
   case BetaNode(_,s1,s2,_) =>
    if (iter(s1) &&  iter(s2)) true else false
   case Leaf(els) => areComplementary(els)
  }

 @tailrec
  def areComplementary(els: List[SF]): Boolean = els match{
   case Nil => false
   case x :: xs =>
    if (xs.exists(p => (x.s != p.s) && (x.f == p.f)))
     true
    else
     areComplementary(xs)
  }

  iter(this)
 }

 override def toString: String = this match {
  case AlphaNode (nte, s, els) =>
   "( Empty ).( " + (nte ::  els(EnumFormula.Alpha)
    ++ els(EnumFormula.Beta) ++
    els(EnumFormula.Var)) + " ).( " + s + " )"
  case BetaNode (nte, l, r, els) =>
   "( " + l + " ).( " + (nte :: els(EnumFormula.Alpha)
    ++ els(EnumFormula.Beta) ++
    els(EnumFormula.Var)) + " ).( " + r + " )"
  case Leaf (els) => els.toString()
 }
}

case class AlphaNode(nodeToExpand: SF, son: Tableaux,
                     els: mutable.Map[EnumFormula.Value, List[SF]])
 extends Tableaux

case class BetaNode(nodeToExpand: SF, left: Tableaux, right: Tableaux,
                    els: mutable.Map[EnumFormula.Value, List[SF]])
 extends Tableaux


case class Leaf(els: List[SF]) extends Tableaux
```

Il Tableaux è stato realizzato definendo un trait Tableaux e delle case class che lo estendono rappresentanti i nodi dell'albero binario.
I nodi dell'albero possono essere di tre tipi:

1) **AlphaNode** => rappresenta il nodo in cui è presente una α - formula da ridurre nel nodo figlio.
2) **BetaNode** => rappresenta il nodo in cui è presente una β – formula da ridurre nei sue due nodi figli.
3) **Leaf** => rappresenta la foglia dell'albero che contiene solo letterali con segno.

L'**AlphaNode** e **BetaNode** sono strutturati contenendo tre informazioni:

1) **NodeToExpand** => identifica la formula segnata (α o β) che deve essere ridotta nei nodi figli.
2) **Left** e **right** => identificano i nodi figli.
3) **Els** => identifica le formule segnate già ridotte o che devono ancora essere ridotte presenti su quel nodo.

Leaf è strutturato contenendo tutte le formule segnate gia ridotte e che non possono essere più ridotte (letterali).

Nel trait **Tableaux** è definito il metodo **isClosed** che verifica se la formula, da cui il tableaux è costruto, è una tautologia. Questo controllo lo fa analizzando le foglie dell'albero. Se in tutte le foglie sono presenti coppie complementari (es: **TX** e **FX**) allora il metodo restituisce true (la formula è una tautologia).

Prover

```
import scala.collection.mutable

class Prover(f: FormulaWff) {

private val sf = SF(F, f)

def prove: Tableaux = {

 def iter(nodeToExpand: SF,
        actualEls: mutable.Map[EnumFormula.Value, List[SF]]):
 Tableaux = {
  if (nodeToExpand == null)
   Leaf(actualEls(EnumFormula.Var))
  else {
   val actualExceptThis = popSF(actualEls, nodeToExpand)
   nodeToExpand match {
    case SF(T, And(l, r)) =>
     AlphaNode(nodeToExpand, iter(nextNodeToExpand(
      pushSF(actualExceptThis,SF(T, l), SF(T, r))),
      pushSF(actualExceptThis,SF(T, l), SF(T, r))),       actualExceptThis)
    case SF(F, Or(l, r)) =>
     AlphaNode(nodeToExpand, iter(nextNodeToExpand(
      pushSF(actualExceptThis,SF(F, l), SF(F, r))),
      pushSF(actualExceptThis,SF(F, l), SF(F, r))), actualExceptThis)
    case SF(F, Implies(l, r)) =>
     AlphaNode(nodeToExpand, iter(nextNodeToExpand(
```

```
        pushSF(actualExceptThis,SF(T, l), SF(F, r))),
        pushSF(actualExceptThis,SF(T, l), SF(F, r))), actualExceptThis)
    case SF(T, Not(s)) =>
     AlphaNode(nodeToExpand, iter(nextNodeToExpand(
       pushSF(actualExceptThis, SF(F, s))),
       pushSF(actualExceptThis, SF(F, s))), actualExceptThis)
    case SF(F, Not(s)) =>
     AlphaNode(nodeToExpand, iter(nextNodeToExpand(
       pushSF(actualExceptThis, SF(T, s))),
       pushSF(actualExceptThis, SF(T, s))), actualExceptThis)
    case SF(F, And(l, r)) =>
     BetaNode(nodeToExpand, iter(nextNodeToExpand(
       pushSF(actualExceptThis, SF(F, l))),
       pushSF(actualExceptThis, SF(F, l))),
       iter(nextNodeToExpand(
         pushSF(actualExceptThis, SF(F, r))),
         pushSF(actualExceptThis,
         SF(F, r))), actualExceptThis)
    case SF(T, Or(l, r)) =>
     BetaNode(nodeToExpand, iter(nextNodeToExpand(
       pushSF(actualExceptThis, SF(T, l))),
       pushSF(actualExceptThis, SF(T, l))),
       iter(nextNodeToExpand(
         pushSF(actualExceptThis, SF(T, r))),
         pushSF(actualExceptThis, SF(T, r))), actualExceptThis)
    case SF(T, Implies(l, r)) =>
     BetaNode(nodeToExpand, iter(nextNodeToExpand(
       pushSF(actualExceptThis, SF(F, l))),
       pushSF(actualExceptThis, SF(F, l))),
       iter(nextNodeToExpand
       (pushSF(actualExceptThis, SF(T, r))),
         pushSF(actualExceptThis, SF(T, r))), actualExceptThis)
    case SF(_, Var(_)) => Leaf(pushSF(actualExceptThis, nodeToExpand)
     (EnumFormula.Var))
   }
  }
}

def popSF(map: mutable.Map[EnumFormula.Value, List[SF]], el: SF) = {
  map(el.getType) = map(el.getType).tail
  map
}

def nextNodeToExpand(map: mutable.Map[EnumFormula.Value, List[SF]]):
  SF = {
  if ((map(EnumFormula.Alpha) == Nil) &&
     (map(EnumFormula.Beta) == Nil))
    null
  else
  if (map(EnumFormula.Alpha) != Nil)
    map(EnumFormula.Alpha).head
  else
    map(EnumFormula.Beta).head
}

def pushSF(map: mutable.Map[EnumFormula.Value, List[SF]], newEls: SF*):
  mutable.Map[EnumFormula.Value, List[SF]] ={
  val temp = mutable.Map(EnumFormula.Alpha -> map(EnumFormula.Alpha),
    EnumFormula.Beta -> map(EnumFormula.Beta),
    EnumFormula.Var -> map(EnumFormula.Var))
```

```
    newEls.foreach(p => temp(p.getType) = p :: temp(p.getType))
    temp
  }

  sf.getType match {
    case EnumFormula.Alpha =>
      iter(sf, mutable.Map(EnumFormula.Alpha -> List(sf),
        EnumFormula.Beta -> Nil, EnumFormula.Var -> Nil))
    case EnumFormula.Beta =>
      iter(sf, mutable.Map(EnumFormula.Alpha -> Nil,
        EnumFormula.Beta -> List(sf), EnumFormula.Var -> Nil))
    case EnumFormula.Var =>
      iter(sf, mutable.Map(EnumFormula.Alpha -> Nil,
        EnumFormula.Beta -> Nil, EnumFormula.Var -> List(sf)))
  }
}

  override def toString: String = "Prover(" + f + ")"

}
```

Il **Prover** è una classe utilizzata per costruire un **Tableaux** a partire da una formula proposizionale. Al suo interno, il **Prover** costruisce una formula segnata, dalla formula proposizionale, con segno F.
A questo punto, con il metodo **prove** costruisco il **Tableaux** analizzando in modo ricorsivo la prossima formula da espandere (partendo dalla radice {FX}). In base al suo operatore logico principale e al suo segno decido quale regola (α o β) utilizzare per costruire il nodo del **Tableaux**. Se applico un' α – regola, costruirò un AlphaNode , se applico una β – regola, costruirò un BetaNode. Il procedimento si ripete sui nodi figli del nodo costruito nel passo precedente, fino a che nelle foglie dell'albero saranno presenti solo letterali con segno.
La scelta della prossima formula da espandere viene effettuata analizzando tutte le formule rimaste fino a quel punto che non sono state ancora elaborate fino alla fine. Prima vengono trattate tutte le α – formule, poi, quando sono terminate, tutte le β – formule.

BuilderFormulaWff

```
import java.io.FileReader

class BuilderFormulaWff(filename: String) {
  private val formula: String = new JTabWbSimpleProblemReader()
    .read(new FileReader(filename)).getConjecture

  private val treeFormula: Formula = {
    if (formula != null)
      new FormulaFactory().buildFrom(new PropositionalFormulaParser()
        .parse(formula))
    else
      null
  }

  def build(): FormulaWff = {
```

```
    def iter(subFormula: Formula): FormulaWff =
      subFormula.getFormulaType match {
      case FormulaType.ATOMIC_WFF =>
        Var(subFormula.toString)
      case FormulaType.AND_WFF =>
        And(iter(subFormula.immediateSubformulas()(0)),
        iter(subFormula.immediateSubformulas()(1)))
      case FormulaType.OR_WFF =>
        Or(iter(subFormula.immediateSubformulas()(0)),
        iter(subFormula.immediateSubformulas()(1)))
      case FormulaType.IMPLIES_WFF =>
        Implies(iter(subFormula.immediateSubformulas()(0)),
        iter(subFormula.immediateSubformulas()(1)))
      case FormulaType.EQ_WFF =>
        And(Implies(iter(subFormula.immediateSubformulas()(0)),
        iter(subFormula.immediateSubformulas()(1))),
        Implies(iter(subFormula.immediateSubformulas()(1)),
          iter(subFormula.immediateSubformulas()(0))))
      case FormulaType.NOT_WFF =>
        Not(iter(subFormula.immediateSubformulas()(0)))
    }

    if (treeFormula == null)
      null
    else
      iter(treeFormula)
  }

}
```

La classe BuilderFormula costruisce, a partire da un file, la rappresentazione ad albero di una formula proposizionale identificata da una stringa.
Il file che viene analizzato deve avere questo formato:

```
*********************************
File     : nome                                  //nome del problema
Status   : provable / unprovable
*********************************
~((~A & B) <=> (~C | D))                         //formula da eseguire
```

Una volta passato il nome del file, BuilderFormula utilizza una libreria Java esterna, scritta dal professore Mauro Ferrari, che verifica la correttezza del file e della formula proposizionale. Dopo aver accertato il formato, la formula viene elaborata e viene costruito una struttura ad albero che la rappresenta.
Ottenuta la struttura, ho definito il metodo **build** che scorre ricorsivamente l'albero e costruisce la formula come albero binario formato dalle classi (quelle relative agli operatori logici) precedentemente descritte. Per quanto riguarda la coimplicazione (<=>), ho deciso di non rappresentarla con un operatore, ma di tradurla in un operazione di congiunzione tra due implicazioni.

ScalaCheck

FormulaSpecification

```
import org.scalacheck.Prop.forAll
import org.scalacheck.{Properties, Test}


object FormulaWffSpecification extends Properties("FormulaWff") {

  property("size(t) <= 2^(depth(t) + 1) - 1") = forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff => f.size <= Math.pow(2, f.depth + 1) - 1
  }

  property("size(t) == size(t1) + size(t2) + 1") = forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff =>
    f match {
      case And(f1, f2) => f.size == f1.size + f2.size + 1
      case Or(f1, f2) => f.size == f1.size + f2.size + 1
      case Implies(f1, f2) => f.size == f1.size + f2.size + 1
      case _ => true
    }

  }

  property("leaves(t) <= 2^(depth(t))") = forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff => f.leaves <= Math.pow(2, f.depth)
  }

  property("leaves(t) == leaves(t1) + leaves(t2)") = forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff =>
    f match {
      case And(f1, f2) => f.leaves == f1.leaves + f2.leaves
      case Or(f1, f2) => f.leaves == f1.leaves + f2.leaves
      case Implies(f1, f2) => f.leaves == f1.leaves + f2.leaves
      case _ => true
    }
  }

  property("depth(t) <= size(t) - 1") = forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff => f.depth <= f.size - 1
  }

  property("depth(t) >= log(2 * size(t))") = forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff => f.depth >= Math.log10(2 * f.size).toInt
  }

  property("depth(t) = max(depth(t1), depth(t2)) + 1") =
forAll(FormulaWffGenerators.myGenFormula) {
    f: FormulaWff => f match {
      case And(f1, f2) => f.depth == Math.max(f1.depth, f2.depth) + 1
      case Or(f1, f2) => f.depth == Math.max(f1.depth, f2.depth) + 1
      case Implies(f1, f2) => f.depth == Math.max(f1.depth, f2.depth) + 1
      case _ => true
```

```
  }
 }

 val checkProp: Seq[(String, Test.Result)] = Test.checkProperties(Test.Parameters.default, this)
}
```

FormulaSpecification è un object che contiene tutte le proprietà da testare con ScalaCheck.
Le proprietà identificate per ogni albero binario sono:

1) Size(t) <= 2^(depth(t) + 1) – 1
2) Size(t) == size(t1) + size(t2) + 1
3) Leaves(t) <= 2^(depth(t))
4) Leaves(t) == leaves(t1) + leaves(t2)
5) Depth(t) <= size(t) – 1
6) Depth(t) >= log(2 * size(t))
7) Depth(t) = max(depth(t1), depth(t2)) + 1

Dove:

1) Size(t) è il numero di nodi dell'albero.
2) Leaves(t) è il numero di foglie dell'albero.
3) Depth(t) è il cammino più lungo dalla radice alle sue foglie.

FormulaGenerators

```
import org.scalacheck.Gen
import org.scalacheck.Gen.oneOf


object FormulaWffGenerators {

 private lazy val genAnd = for {
  left <- myGenFormula
  right <- myGenFormula

 } yield And(left, right)


 private lazy val genOr = for {
  left <- myGenFormula
  right <- myGenFormula

 } yield Or(left, right)


 private lazy val genImplies = for {
  left <- myGenFormula
  right <- myGenFormula

 } yield Implies(left, right)


 private val genNot = for {
  son <- myGenFormula
```

```
  } yield Not(son)


  private lazy val genVar = oneOf(Var("A"),Var("B"))

  def myGenFormula: Gen[FormulaWff] =
    Gen.frequency(
      4 -> genVar,
      1 -> Gen.lzy(genAnd),
      1 -> Gen.lzy(genOr),
      1 -> Gen.lzy(genImplies),
      1 -> Gen.lzy(genNot)
    )
}
```

FormulaGenerators è un object che contiene tutti i generatori usati da ScalaCheck per verificare le proprietà descritte precedentemente. La soluzione adottata per la definizione del generatore myGenFormula consiste nel forzare a generare una formula proposizionale con probabilità di ottenere una variabile più alta rispetto a tutti gli altri operatori logici. In questo modo si evita l'errore StackOverflow causato dalla creazione ricorsiva di formule senza terminazione.

Ringraziamenti

Per la redazione di questo lavoro ci terrei a ringraziare prima di tutto il professore e relatore Mauro Ferrari che mi ha dedicato il suo tempo e mi ha dato la possibilità di poter essere qui oggi.

Ringrazio la mia famiglia, in particolare mia madre, che in questi anni mi ha supportato e sopportato. Ha creduto in me quando a nessun'altro importava farlo.

Ringrazio infine l'Università dell'Insubria e tutti i professori che ogni giorno trasmettono il loro sapere agli studenti.

Sitografia

1) https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md
2) https://github.com/ferram/jtabwb

Bibliografia

1) Paul Chiusano e Rúnar Bjarnason, Functional programming in Scala, Manning Publications, 2014
2) Raymond M. Smullyan, First - Order Logic, Dover Pubns, 1995