Sistema certificado de decisión proposicional basado en polinomios

***[1]

Universidad de Sevilla.
Dpto. Ciencias de la Computación e Inteligencia Artificial
ETS Ingeniería Informática. Sevilla , Spain

Abstract. En [citar calculemus] hemos introducido un procedimiento de decisión proposicional basado en el uso de la derivación polinomial. En el presente trabajo presentamos una implementación en Haskell del procedimiento y su certificación mediante QuickCheck.

1 Introducción

Este trabajo puede considerarse como un trabajo de tránsito: es la implementación en Haskell y certificación con QuickCheck del procedimiento de decisión introducido en [calculemus] y su objetivo es la verificación formal de la implementación en Isabelle. Sólo presentamos los elementos más destacados, el código completo puede consultarse en `CLAI2009.hs`

2 Lógica proposicional

Esta sección se describen la implementación en Haskell de los conceptos de la lógica proposicional necesarios para la certificación de nuestro procedimiento de decisión.

2.1 Gramática de la lógica proposicional

Los símbolos proposicionales se representarán mediante cadenas.

```
type SimboloProposicional = String
```

Las fórmulas proposicionales se definen por recursión:

- $\top$ y $\bot$ son fórmulas
- Si A es una fórmula, también lo es $\neg A$.
- Si A y B son fórmulas, entonces $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ y $(A \leftrightarrow B)$ también lo son.

```
data FProp = Atom SimboloProposicional
           | T
           | F
           | Neg FProp
           | Conj FProp FProp
           | Disj FProp FProp
           | Impl FProp FProp
           | Equi FProp FProp
           deriving (Eq,Ord)
```

Para facilitar la escritura de fórmulas definimos las fórmulas atómicas `p`, `q`, `r` y `s` así como las funciones para las conectivas `no`, `/\`, `\/`, `-->` y `<-->`. Asímimo, se define el procedimiento `show` para facilitar la lectura de fórmulas y el procedimiento `arbitrary` para generar fórmulas aleatoriamente.

2.2 Semántica de la lógica proposicional

Las interpretaciones las representamos como listas de fórmulas, entendiendo que las fórmulas verdaderas son las que pertenecen a la lista.

```
type Interpretacion = [FProp]
```

Por recursión, se define la función `(significado f i)` que es el significado de la fórmula `f` en la interprestación `i` y a partir de ella las funciones para reconocer si una interpretación es modelo de una fórmula o de un conjunto de fórmulas.

```
esModeloFormula :: Interpretacion -> FProp -> Bool
esModeloFormula i f = significado f i

esModeloConjunto :: Interpretacion -> [FProp] -> Bool
esModeloConjunto i s =
    and [esModeloFormula i f | f <- s]
```

Por recursión se define la función `(simbolosPropForm f)` que es el conjunto formado por todos los símbolos proposicionales que aparecen en la fórmula `f`. A partir de ella, se definen las funciones para generar las interpretaciones y los modelos de una fórmula

```
interpretacionesForm :: FProp -> [Interpretacion]
interpretacionesForm f = subconjuntos (simbolosPropForm f)

modelosFormula :: FProp -> [Interpretacion]
modelosFormula f =
    [i | i <- interpretacionesForm f, esModeloFormula i f]
```

Análogamente se generan los modelos de conjuntos de fórmulas

```
modelosConjunto :: [FProp] -> [Interpretacion]
modelosConjunto s =
    [i | i <- interpretacionesConjunto s, esModeloConjunto i s]

interpretacionesConjunto :: [FProp] -> [Interpretacion]
interpretacionesConjunto s = subconjuntos (simbolosPropConj s)
```

donde `(simbolosPropConj s)` es el conjunto de los símbolos proposiciones del conjunto de fórmulas `s`.

Finalmente se definen las funciones para reconocer si un conjunto de fórmulas es inconsistente, si una fórmula es consecuencia de un conjunto de fórmulas, si una fórmula es válida y si dos fórmulas son equivalentes:

```
esInconsistente :: [FProp] -> Bool
esInconsistente s =
    modelosConjunto s == []

esConsecuencia :: [FProp] -> FProp -> Bool
esConsecuencia s f = esInconsistente (Neg f:s)

esValida :: FProp -> Bool
esValida f = esConsecuencia [] f

equivalentes :: FProp -> FProp -> Bool
equivalentes f g = esValida (f <--> g)
```

3 Polinomios de $\mathbb{Z}/\mathbb{Z}_2$

3.1 Representación de polinomios

Las variables se representan por cadenas. Los monomios son productos de variables y los representamos por listas ordenadas de variables. Los polinomios son suma de monomios y los representamos por listas ordenadas de monomios.

```
type Variable = String
data Monomio = M [Variable] deriving (Eq, Ord)
data Polinomio = P [Monomio] deriving (Eq, Ord)
```

Los reconocedores de monomios y polinomios se definen por

```
esMonomio :: Monomio -> Bool
esMonomio (M vs) = vs == sort (nub vs)

esPolinomio :: Polinomio -> Bool
esPolinomio (P ms) = ms == sort (nub ms)
```

El monomio correspondiente a la lista vacía es el monomio uno, el polinomio correspondiente a la lista vacía es el polinomio cero y el correspondiente a la lista cuyo único elemento es el monomio uno es el polinomio uno,

```
mUno :: Monomio
mUno = M []

cero, uno  :: Polinomio
cero = P []
uno = P [mUno]
```

3.2 Operaciones con polinomios

Las definiciones de suma de polinomios y producto de polinomios son

```
suma :: Polinomio -> Polinomio -> Polinomio
suma (P []) (P q) = (P q)
suma (P p)  (P q) = P (sumaAux p q)
   where sumaAux [] y      = y
         sumaAux (c:r) y
             | elem c y   = sumaAux r (delete c y)
             | otherwise  = insert c (sumaAux r y)

producto :: Polinomio -> Polinomio -> Polinomio
producto (P []) _     = P []
producto (P (m:ms)) q = suma (productoMP m q) (producto (P ms) q)

productoMP :: Monomio -> Polinomio -> Polinomio
productoMP m1 (P [])     = P []
productoMP m1 (P (m:ms)) =
   suma (P [productoMM m1 m]) (productoMP m1 (P ms))

productoMM :: Monomio -> Monomio -> Monomio
productoMM (M x) (M y) = M (sort (x `union` y))
```

3.3 Generación de polinomios y certificación con QuickCheck

Para generar aleatoriamente polinomios se definen generadores de caracteres de letras minúsculas, de variables con longitud 1 ó 2, de monomios con el número de variables entre 0 y 3 y de polinomios con el número de monomios entre 0 y 3.

```
caracteres :: Gen Char
caracteres = chr `fmap` choose (97,122)

variables :: Gen String
```

```
variables = do k <- choose (1,2)
               vectorOf k caracteres

monomios :: Gen Monomio
monomios = do k <- choose (0,3)
              vs <- vectorOf k variables
              return (M (sort (nub vs)))

polinomios :: Gen Polinomio
polinomios = do k <- choose (0,3)
                ms <- vectorOf k monomios
                return (P (sort (nub ms)))
```

Por ejemplo,

```
*Main> sample polinomios
0
pp*sa
gn*nf*zg
```

Declaramos los polinomios instancias de Arbitrary, para usar QuickCheck, y de Num para facilitar la escritura.

```
instance Arbitrary Polinomio where
    arbitrary = polinomios

instance Num Polinomio where
    (+) = suma
    (*) = producto
    (-) = suma
    abs = undefined
    signum = undefined
    fromInteger = undefined
```

La propiedades de las operaciones sobre polinomios se certifican con QuickCheck

```
prop_polinomios :: Polinomio -> Polinomio -> Polinomio -> Bool
prop_polinomios p q r =
   esPolinomio (p+q)       &&
   p+q == q+p              &&
   p+(q+r) == (p+q)+r      &&
   p + cero == p           &&
   p+p == cero             &&
   esPolinomio (p*q)       &&
   p*q == q*p              &&
   p*(q*r) == (p*q)*r      &&
   p * uno == p            &&
```

```
   p*p == p                &&
   p*(q+r) == (p*q)+(p*r)
```

4 Polinomios y fórmulas proposicionales

Los polinomios pueden transformarse en fórmulas y las fórmulas en polinomios de manera que las transformaciones sean reversibles.

```
tr :: FProp -> Polinomio
tr T          = uno
tr F          = cero
tr (Atom p)   = P [M [p]]
tr (Neg p)    = uno + (tr p)
tr (Conj a b) = (tr a) * (tr b)
tr (Disj a b) = (tr a) + (tr b) + ((tr a) * (tr b))
tr (Impl a b) = uno + ((tr a) + ((tr a) * (tr b)))
tr (Equi a b) = uno + ((tr a) + (tr b))

theta :: Polinomio -> FProp
theta (P ms) = thetaAux ms
   where thetaAux []     = F
         thetaAux [m]    = theta2 m
         thetaAux (m:ms) = no ((theta2 m) <--> (theta (P ms)))
```

Las propiedades de reversibilidad pueden comprobarse con QuickCheck

```
prop_theta_tr :: FProp -> Bool
prop_theta_tr p = equivalentes (theta (tr p)) p

prop_tr_theta :: Polinomio -> Bool
prop_tr_theta p = tr (theta p) == p
```

5 Procedimiento de decisión proposicional basado en polinomios

5.1 Derivación y regla delta

La derivada de un polinomio p respecto de una variable x es la lista de monomios p que contienen x, eliminándola. La derivación se extiende a las fórmulas mediante las funciones de transformación

```
deriv :: Polinomio -> Variable -> Polinomio
deriv (P ms) x = P [M (delete x m) | (M m) <- ms, elem x m]
```

```
derivP :: FProp -> Variable -> FProp
derivP f v = theta (deriv (tr f) v)
```

La regla delta para polinomios y fórmula es

```
deltap :: Polinomio -> Polinomio -> Variable -> Polinomio
deltap a1 a2 v = uno + ((uno+a1*a2)*(uno+a1*c2+a2*c1+c1*c2))
    where
      c1 = deriv a1 v
      c2 = deriv a2 v

delta :: FProp -> FProp -> Variable -> FProp
delta f1 f2 v = theta (deltap (tr f1) (tr f2) v)
```

Con QuickCheck se comprueba que la regla delta es adecuada

```
prop_adecuacion_delta :: FProp -> FProp -> Bool
prop_adecuacion_delta f1 f2 =
    and [esConsecuencia [f1, f2] (delta f1 f2 x) |
         x <- variablesProp (f1 /\ f2)

variablesProp f = [v | (Atom v) <- simbolosPropForm f]
```

5.2 Procedimiento de decisión

La función `(derivadas fs x)` es la lista de las proposiciones distintas de $\top$ obtenidas aplicando la regla delta a dos fórmulas de `fs` respecto de la variable `x`.

```
derivadas :: [FProp] -> Variable -> [FProp]
derivadas fs x =
   delete T (nub [delta f1 f2 x | (f1,f2) <- pares fs])

pares :: [a] -> [(a,a)]
pares []  = []
pares [x] = [(x,x)]
pares (x:xs) = [(x,y) | y <- (x:xs)] ++ (pares xs)
```

A partir de la anterior, se determina cuando un conjunto de fórmulas es refutable por la regla delta

```
deltaRefutable :: [FProp] -> Bool
deltaRefutable [] = False
deltaRefutable fs =
    (elem F fs) ||
```

```
    (deltaRefutable (derivadas fs (eligeVariable fs)))
    where eligeVariable fs =
             head (concat [variablesProp f | f <- fs])
```

Con QuickCheck se comprueba que el procedimiento de delta–refutación es adecuado y completo

```
prop_adecuacion_completitud_delta :: [FProp] -> Bool
prop_adecuacion_completitud_delta fs =
    esInconsistente fs == deltaRefutable fs
```

A partir del procedimiento de refutación se pueden definir los de demostrabilidad y validez

```
deltaDemostrable :: [FProp] -> FProp -> Bool
deltaDemostrable fs g = deltaRefutable ((no g):fs)

prop_adecuacion_completitud_delta_demostrable :: [FProp] ->
                                                 FProp ->
                                                 Bool
prop_adecuacion_completitud_delta_demostrable fs g =
    esConsecuencia fs g == deltaDemostrable fs g

deltaTeorema :: FProp -> Bool
deltaTeorema f = deltaRefutable [no f]

prop_adecuacion_completitud_delta_teorema :: FProp -> Bool
prop_adecuacion_completitud_delta_teorema f =
    esValida f == deltaTeorema f
```

6 Conclusiones y trabajos futuros

Este trabajo puede considerarse como un trabajo de tránsito: es la implementación en Haskell y certificación con QuickCheck del procedimiento de decisión introducido en [calculemus] y su objetivo es la verificación formal de la implementación en Isabelle