Sistema certificado de decisión proposicional basado en polinomios

José A. Alonso-Jiménez, Gonzalo A. Aranda-Corral y Joaquín Borrego-Díaz

Dpto. Ciencias de la Computación e Inteligencia Artificial
ETS Ingeniería Informática - Universidad de Sevilla.
Avda. Reina Mercedes s.n.
41012-Sevilla , Spain
{jalonso, jborrego, garanda} @us.es

Abstract. En [2] introducimos una regla de inferencia para la lógica proposicional (basado en el uso de la derivación de polinomios) denominada *regla de independencia*, diseñada para el cálculo de retracciones conservativas, y que sirve para diseñar un procedimiento de decisión proposicional. 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[7] y certificación con QuickCheck[4] del procedimiento de decisión introducido en [2] y su objetivo es la verificación formal de la implementación en el sistema Isabelle. Sólo presentamos los elementos más destacados, el código completo puede consultarse en `CLAI2009.hs`.

La regla de independencia, en la que se basa el procedimiento de decisión, se basa a su vez en una interpretación lógica de la derivación polinomial en característica 2. La idea de un operador de derivación sobre fórmulas proposicionales consiste en la traducción de las derivaciones usuales para aplicarlas sobre fórmulas vía la traducción clásica entre fórmulas y polinomios sobre cuerpos finitos [5] (véase también [3]). Para más detalles sobre la definición formal y sus propiedades se puede consultar [2]. En [1] se aplica la derivación y la regla para extender los métodos clásicos de exploración de atributos en contextos formales.

La derivada booleana es una herramienta muy utilizada en el cálculo de funciones booleanas (cf. [8]). En nuestro, caso, el operador sobre fórmulas se obtiene como una traducción de los operadores usuales de derivación sobre anillos, es decir, operadores sobre un anillo R $d : R \to R$ verificando que:

1. $d(a + b) = d(a) + d(b)$
2. $d(a \cdot b) = d(a) \cdot b + a \cdot d(b)$

Una *derivación booleana* es una aplicación $\partial : PForm \to PForm$ tal que existe una derivación d sobre $\mathbb{F}_2[\mathbf{x}]$ tal que el siguiente diagrama es conmutativo

$$\begin{array}{ccc} PForm & \stackrel{\partial}{\to} & PForm \\ \pi \downarrow & \# & \uparrow \Theta \\ \mathbb{F}_2[\mathbf{x}] & \stackrel{d}{\to} & \mathbb{F}_2[\mathbf{x}] \end{array}$$

(donde π y Θ son las transformaciones usuales entre polinomios y fórmulas proposicionales. Es decir

$$\partial = \Theta \circ d \circ \pi$$

Si $d = \frac{\partial}{\partial x_p}$, se denota ∂ por $\frac{\partial}{\partial p}$.

La fórmula $\frac{\partial}{\partial p}(F)$ expresa la dependencia del valor de F con respecto al valor de p, es decir,

$\frac{\partial}{\partial p}F \equiv \neg(F\{p/\neg p\} \leftrightarrow F)$.

La regla de independecia (o regla ∂) sobre dos fórmulas polinomiales (polinomios de grado a lo sumo 1 con respecto a cualquier variable) $a_1, a_2 \in \mathbb{F}_2[\mathbf{x}]$ se define como:

$$\frac{a_1,\ a_2}{1 + \Phi\left[(1 + a_1 \cdot a_2)(1 + a_1 \cdot \frac{\partial}{\partial x}a_2 + a_2 \cdot \frac{\partial}{\partial x}a_1 + \frac{\partial}{\partial x}a_1 \cdot \frac{\partial}{\partial x}a_2)\right]}$$

donde

$$\Phi(\sum_{\alpha \in I} x^\alpha) = \sum_{\alpha \in I} x^{sg(\alpha)} \text{ y } sg(\alpha_1, \ldots, \alpha_n) = (\min(1, \delta_1), \ldots, \min(1, \delta_n))$$

Al resultado lo denotamos por $\partial_x(a_1, a_2)$. Es decir, si $a_i = b_i + x_p \cdot c_i$, con $\deg_{x_p}(b_i) = \deg_{x_p}(c_i) = 0$ $(i = 1, 2)$,

$$\partial_x(b_1 + x_p \cdot c_1,\ b_2 + x_p \cdot c_2) = \Phi\left[1 + (1 + b_1 \cdot b_2)[1 + (b_1 + c_1)(b_2 + c_2)]\right]$$

La regla de independencia sobre fórmulas es:

$$\partial_p(F_1, F_2) := \Theta(\partial_{x_p}(\pi(F_1), \pi(F_2))).$$

El cálculo inducido por la regla de independencia es adecuado y refutacionalmente completo [2]. La prueba de la completitud muestra, de hecho, un procedimiento de decisión basado en la regla que consiste en la retracción conservativa del conjunto de fórmulas a analizar, eliminando una a una las variables proposicionales: Si Γ es un conjunto de fórmulas inconsistente, entonces $\Gamma \vdash_\partial \bot$.

La prueba consiste en calcular los conjuntos $\partial_k[\Gamma]$ $(k \leq n)$ definidios por recursion como sigue: $\partial_0[\Gamma] := \Gamma$ and, if $k \geq 1$,

$$\partial_k[\Gamma] := \{\partial_{p_k}(F_1, F_2)\ :\ F_1, F_2 \in \partial_{k-1}[\Gamma]\}$$

Nótese que si $F \in \partial_k[\Gamma]$, entonces $\mathrm{var}(F) \subseteq \{p_{k+1}, \ldots p_n\}$. Por tanto, $\partial_n[\Gamma] \subseteq \{\top, \bot\}$. Se verifica que Γ is inconsistente si y sólo si $\bot \in \partial_n(\Gamma)$.

2 Formalización de la 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 y 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 $(A \wedge B)$, $(A \vee B)$, $(A \to B)$ y $(A \leftrightarrow B)$ también lo son.

mediante los constructores T, F, Neg, Conj, Disj, Impl y Equiv, respectivamente. Para facilitar su escritura definimos las fórmulas atómicas `p`, `q`, `r` y `s` así como las funciones para las conectivas `no`, `/\`, `\/`, `-->` y `<-->`. Asímismo, 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 Formalización del álgebra de polinomios sobre $\mathbb{Z}/2\mathbb{Z}$

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

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
```

3.4 Certificación con QuickCheck

A continuación vamos a usar la herramienta QuickCheck para la certificación de la implementación realizada de los polinomios, y posteriormente de los resultados que se obtengan también con fórmulas proposicionales.

QuickCheck es una herramienta que nos permite definir propiedades de los programas, que suponemos que son ciertas, y testearlas con un gran número de ejemplos. Fue desarrollado por Claessen and Hughes para Haskell. Los puntos fuertes de esta herramienta son la facilidad para la escritura de propiedades y la rapidez de comprobación, siendo para ello muy importante la definición correcta de los generadores de ejemplos.

Además, el uso de QuickCheck para la certificación de nuestro programa es un paso previo a su verificación formal con Isabelle. Recientemente se ha creado una herramienta de traducción automática desde código Haskell a Isabelle, denominada Haskabelle[1], pero que se encuentra en fase de desarrollo, y que pensamos usar para convertir nuestro código en una teoría y poder demostrar formalmente sus propiedades.

En el caso concreto de los polinomios certificaremos todas las operaciones implementadas exigiendo que *estén bien definidas* y que cumplan las propiedades, que en sintaxis de QuickCheck se escribe de la siguiente forma:

```
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]]
```

[1] http://isabelle.in.tum.de/haskabelle

```
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 [2] y su objetivo es la verificación formal de la implementación en Isabelle. Otra línea que consideramos interesante es el refinamiento del algoritmo de decisión presentado (basado en la saturación del conjunto por aplicación de la regla) para obtener versiones del algoritmo más eficientes y certificados para el problema SAT.

References

1. J. A. Alonso-Jiménez, G. A. Aranda-Corral, J. Borrego-Díaz, M. M. Fernández-Lebrón and M. J. Hidalgo-Doblado: *Extending Attribute Exploration by Means of Boolean Derivatives* Proc. of 6th International Conference Concept Lattices and Their Applications (CLA 2008), pp.121-132 (2008) http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-433/
2. G. A. Aranda-Corral, J. Borrego-Díaz and M. M. Fernández-Lebrón, *Conservative Retractions of Propositional Logic Theories by Means of Boolean Derivatives: Theoretical Foundations*, Proc. Calculemus 2009, Lecture Notes in Artificial Intelligence n. 5625. pp: 45-58, Springer (2009).
3. J. Chazarain, J. A. Alonso-Jiménez, E. Briales-Morales and A. Riscos-Fernández, Multi-valued logic and Gröbner bases with applications to modal logic. *Journal Symbolic Computation* **11** 181–194 (1991).
4. Koen Claessen and John Hughes, QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, *ACM SIGPLAN Notices* (ACM Press): 268–279, 2000
5. kapur D. Kapur and P. Narendran, An equational approach to theorem proving in first-order predicate calculus, *Proc. 9 Int. Joint Conf. on Artificial Intelligence (IJCAI'85)*, 1146-1153.

6. L. M. Laita, E. Roanes-Lozano, L. de Ledesma and J. A. Alonso-Jiménez. A computer algebra approach to verification and deduction in many-valued knowledge systems. *Soft Computing* **3**: 7–19 (1999).
7. Simon Peyton Jones and others, The Haskell 98 Language and Libraries: The Revised Report, *Journal of Functional Programming* **13**: 0–255,(20033)
8. A. Thayse. *Boolean Calculus of Differences*. Springer, Berlin, 1981.