Programación con Tipos Inductivos *

J. L. Freire Nistal e A. Blanco Ferro

26 de xaneiro de 1999

Palabras clave: *Proba automática, Teoría de tipos*

1 Introducción

Neste traballo presentarase unha maneira de construir programas sobre tipos de datos definidos inductivamente. Esta forma de introducir os tipos de datos só é posible nun contexto onde convivan tanto as costruccións de novos obxectos (parte conxuntista) como as propiedades destes obxectos (parte proposicional).

Un destes contextos é **Coq**, unha implementación do Cálculo de Construccións inductivas (CIC) de Gérard Huet, feita no INRIA por [Dowek et al]. Este sistema é un probador automático de teoremas dirixido por obxectivos (*goal-directed*) e por tácticas (*tactic-driven*) no estilo do LCF [Gordon et al], que permite a verificación formal de programas e, mesmo, a sú a xeración automática. Tamén ten soporte automático para a búsqueda de probas así como para definir tipos inductivamente de xeito que, constructores de probas inductivas e mesmo funcións, defínense automáticamente.

O noso obxectivo consiste e amosar a potencia do constructor de programas asociado a un tipo Inductivo, e utilizala para definir programas directamente.

Tamén esperamos que esta reflexión sobre á ferramenta Coq sirva para despertar o interés por os probadores automáticos.

2 Unha breve descripción de Coq

CIC é unha teoría de tipos que resulta da combinación da teoría intuicionista de tipos de Martin-Löf e o λ–cálculo polimórfico de Girard F_ω. Os teoremas a probar se representan como tipos, e as probas son termos que teñan ese tipo. Este xeito de ver as cousas e consecuencia do isomorfismo de Curry-Howard [Howard]

Hay dous tipos básicos en Coq: "Set" para definición de obxectos, e "Prop" que permite a definición de predicados e relacións que atinxen a estes obxectos, así como proposicións a probar sobre eles.

*Supported by Xunta de Galicia XUGA10504B96

A notación `a:A` leerase, `a` é de tipo `A`, e interpretarase como `a` é unha proba de "A", si `A` é de tipo "Prop" ou ben `a` é un elemento de `A`, si `A` é de tipo "Set".

As construccións permitidas son: `x|(M N)|[x:T]M|(x:T)P` . Na primeira `x` denota variables ou constantes; a segunda é a aplicación. A terceira representa o programa (λ–expresión) de parámetro `x` e corpo `M`. E finalmente, a cuarta é o tipo dos programas que admiten unha entrada de tipo `T` e devolven un resultado de tipo `P`. Este tipo chámase tamén tipo producto e en teoría de tipos represéntase por $\Pi x : T.P$ ou tamén $\forall x : T.P$. Si `x` non figura libre en `P` entón escríbese tamén `T -> P` que é o tipo das funcións entre esos dous tipos.

En Coq pódense definir tipos inductivos. Por exemplo, para definir os números naturais `nat`, a definición:

```
Coq < Inductive nat : Set := O : nat | S : nat->nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined
```

engade o tipo `nat` e os constructores `O` e `S` ó entorno.

```
Coq < Check nat.
nat
     : Set
Coq < Check O.
O
     : nat
Coq < Check S.
S
     : nat->nat
```

Aparecen ademais os termos novos `nat_ind` e `nat_rec` cuios valores e tipos podemos chequear co sistema:[1]

```
Coq < Print nat_ind.
nat_ind =
[P:nat->Prop]
 [f:(P O)]
  [f0:(n:nat)(P n)->(P (S n))]
   Fix F
     {F [n:nat] : (P n) :=
        Cases n of
          O => f
        | (S n0) => (f0 n0 (F n0))
        end}
     : (P:nat->Prop)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)


Coq < Print nat_rec.
```

[1] o terceiro termo é análogo os outros dous pero referido o tipo `Type`

```
nat_rec =
[P:nat->Set]
 [f:(P O)]
  [f0:(n:nat)(P n)->(P (S n))]
   Fix F
     {F [n:nat] : (P n) :=
        Cases n of
          O => f
        | (S n0) => (f0 n0 (F n0))
        end}
     : (P:nat->Set)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

o primeiro usarase para facer probas por inducción de proposicións `P:nat->Prop`. O que dí non é mais que o ben coñecido principio de inducción: para ter unha proba de $\forall n : nat.(P(n))$ (é decir, para construir un elemento de tipo `(n:nat)(P n)`) bastará ter unha proba de `(P O)` e un procedemento de tipo `(n:nat)(P n)->(P (S n))` (é decir, unha proba de $\forall n : nat.(P(n)) \Rightarrow (P(S(n))))$.

Por exemplo:

```
Coq < Variable predicado:nat -> Prop.
predicado is assumed
```

así declárase unha variable `predicado`

```
Coq < Lemma foo:(n:nat)(predicado n).
1 subgoal

  ============================
   (n:nat)(predicado n)
```

desta maneira enúnciase un lema que haberá que demostrar usando o sistema interactivamente mediante tácticas. Éstas aparecen como comandos do sistema dentro deste nivel, o cual ven sinalado por o distinto *prompt* que aparece a partires deste momento `foo`

```
foo < Intro.
1 subgoal

  n : nat
  ============================
   (predicado n)
```

esta táctica engade `n:nat` ó entorno das hipótesis e deixa como subobxectivo `(predicado n)`.

```
foo < Elim n.
2 subgoals

  n : nat
```

```
  ============================
   (predicado O)

subgoal 2 is:
 (n:nat)(predicado n)->(predicado (S n))
```

o uso da táctica `Elim` permite a utilización de `nat_ind`. Nun exemplo mais concreto habería un predicado particular e a partires del seguiría a demostración.

O segundo (`nat_rec`) expresa que si temos unha familia P_n de conxuntos e queremos definir unha maneira de elexir un elemento en cada un deles (é decir, un elemento de $\Pi x : T.P$), entón bastará ter un elemento de P_O e un procedemento de tipo `(n:nat)(P n)->(P (S n))`.

Por exemplo:

```
Coq < Definition Suma :=
       ( nat_rec
                 ([_:nat]nat->nat)
                 ([n:nat]n)
                 ([_:nat][f:nat->nat][x:nat](S (f x)))
        ).
Suma is defined
```

Podemos definir ahora inductivamente unha proposición con tres variables que expresa o feito de que o terceiro número é a suma do primeiro e o segundo. Nótese que os "constructores" neste caso expresan axiomas.

```
Coq < Inductive suma:nat->nat->nat->Prop:=
      suma0:(m:nat)(suma O m m)
    | sumaS:(m,n,p:nat)(suma m n p)->(suma (S m) n (S p)).
```

e agora verificar que o noso programa `Suma` é correcto:

```
Coq < Theorem testsuma:(m,n:nat)(suma m n (Suma m n)).
Coq < Intros.
Coq < Elim m.
Coq < Simpl.
Coq < Apply suma0.
Coq < Intros.
Coq < Simpl.
Coq < Apply sumaS.
Coq < Assumption.

Subtree Proved!
```

Xa que logo, o tipo `Prop` resérvase para definir proposicións, aseveracións que son susceptibles de ser probadas. Os predicados e as relacións se definen como termos funcionais de conxuntos a "Prop". Por exemplo, un predicado sobre os números naturais terá tipo `nat->Prop`, e unha relación binaria sobre `nat`, terá tipo `nat->nat->Prop`.

En Coq, algúns conectivos son primitivos, mentras que outros son definidos a partir destes. As súas definicións e reglas de inferencia correspondentes se cargan ó arranque do sistema. Por exemplo, o tipo producto `(x:T)P` pode verse como unha cuantificación universal sobre `x`. O caso particular `T->P` cando o `x` non aparece libre en `P`, correspóndese con a implicación.

```
Coq < Inductive True : Prop :=  I : True.
Coq < Print True_ind.
True_ind =
[P:Prop][f:P][t:True]Cases t of I => f end
     : (P:Prop)P->True->P
```

o principio de inducción neste caso significa que si se quere probar $True \Rightarrow P$, basta probar P.

```
Coq < Inductive False : Prop :=
Coq < Print False_ind.
False_ind = [P:Prop][f:False]<P>Cases f of end
     : (P:Prop)False->P
```

o principio de inducción neste caso significa que si se quere probar $False \Rightarrow P$, non fai falta nada!

```
Coq < Print and.
Inductive and [A:Prop; B:Prop] : Prop :=  conj : A->B->(and A B)
Coq < Print and_ind.
and_ind =
[A,B,P:Prop][f:A->B->P][a:A/\B]Cases a of (conj x x0) => (f x x0)
     : (A,B,P:Prop)(A->B->P)->A/\B->P
```

aquí o que temos como principio de inducción asociado, é que si queremos probar que $A \wedge B \Rightarrow P$ entón bastará probar $A \Rightarrow B \Rightarrow P$. Por exemplo:

```
Coq < Lemma andcom:(A,B:Prop)(A/\B)->(B/\A).
1 subgoal

  ============================
   (A,B:Prop)A/\B->B/\A
andcom < Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  ============================
   B/\A
andcom < Elim H.
1 subgoal
```

```
  A : Prop
  B : Prop
  H : A/\B
  ============================
   A->B->B/\A
```

esta táctica `Elim` aplica a inducción de `and`. En xeral a eliminación dunha hipótese mediante `Elim` aplica o principio de inducción correspondente.

```
andcom < Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
  ============================
   B/\A
andcom < Split.
2 subgoals

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
  ============================
   B

subgoal 2 is:
 A
```

a táctica `Split` aplica o feito (axioma) `(conj B A)` que é unha proba de `B->A->(B/\A)`. Expresa que para ter unha proba de `B/\A` bastará ter `b:B` e `a:A`, pois entón `(conj B A b a):B/\A`.

```
andcom < Assumption.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
  ============================
   A

andcom < Assumption.
```

```
Subtree proved!
andcom < Save.
Intros.
Elim H.
Intros.
Split.
Assumption.

Assumption.

andcom is defined
```

Do mesmo xeito se teñen `~P` para a negación, `A\/B`, `Ex([x:T]P)` para a cuantificación existencial, `a=b` para a igualdade, etc.

Terase notado o carácter estrictamente constructivo das probas en Coq, cuia lóxica é intuicionista.

3 Algúns exemplos

Os exemplos que siguen tratan de ilustrar a idea de que a definición dos algoritmos sobre un tipo dado pódese sistematizar co uso do constructor recursivo asociado. Tamén veremos como nestes exemplos é doado facer a verificación da corrección.

3.1 O Producto Cartesiano

Consideremos ahora o tipo producto de conxuntos:

```
Coq < Inductive prod [A:Set; B:Set] : Set :=  pair : A->B->(prod A B)
prod_ind is defined
prod_rec is defined
prod_rect is defined
prod is defined
Coq < Print prod_rec.
prod_rec =
[A,B:Set]
 [P:(prod A B)->Set]
  [f:(y:A)(y0:B)(P (pair A B y y0))]
   [p:(prod A B)]Cases p of (pair x x0) => (f x x0) end
     : (A,B:Set)
        (P:(prod A B)->Set)
         ((y:A)(y0:B)(P (pair A B y y0)))->(p:(prod A B))(P p)
Coq < Print prod_ind.
prod_ind =
[A,B:Set]
 [P:(prod A B)->Prop]
  [f:(y:A)(y0:B)(P (pair A B y y0))]
   [p:(prod A B)]Cases p of (pair x x0) => (f x x0) end
     : (A,B:Set)
```

```
(P:(prod A B)->Prop)
 ((y:A)(y0:B)(P (pair A B y y0)))->(p:(prod A B))(P p)
```

aquí o principio de inducción non é moi interesante. Centrémonos no principio de recursividade que define `prod_rec` Por exemplo. Consideremos a función especificada do seguinte modo: a funcion $\pi_1 : A \times B \to A$ definida por $\pi_1(x,y) = x$. Podémola obter con `prod_rec` pasándolle como parámetros: $P : A \times B \to Set$ que será `[_:A*B]A` (a función que a calquera cousa de tipo `A*B` asocialle `A`; despois o segundo argumento ten que ser de tipo $\forall y : A.\forall y_0 : B.A$ pois P é a constante `A`, é decir, de tipo `A->B->A`. Esa función debe ser a que a calquera `x:A` e a calquera cousa de tipo `B` lle asocie xustamente o `x`.

```
Coq < Definition pi_1:(A,B:Set)A*B->A :=
Coq <       [A:Set][B:Set](prod_rec A B [_:A*B]A [x:A][_:B]x).
pi_1 is defined
Coq < Check pi_1.
pi_1
     : (A,B:Set)A*B->A
```

e usando o comando `Eval Compute in`, que leva a cabo as reduccións necesarias, é fácil ver que `pi_1` cumple a especificación :

```
Coq < Variables A,B:Set. Variable a:A. Variable b:B.
Coq < Eval Compute in (pi_1 A B (a,b)).
     = a
     : A
```

En xeral, si se quere definir un programa de tipo `(A,B,C:Set)A*B->C` entón o esquema será:

```
Coq < Definition fooprod := [A:Set][B:Set][C:Set](prod_rec A B [_:A*B]C).
Coq < Check fooprod.
fooprod
     : (A,B,C:Set)((y:A)(y0:B)([_:A*B]C (y,y0)))->(p:A*B)([_:A*B]C p)
```

este resultado pódese simplificar usando o comando `Eval Simpl in`

```
Coq < Eval Simpl in fooprod.
     = fooprod
     : (A,B,C:Set)(A->B->C)->A*B->C
```

xa que logo, aplicando `foo` a unha función de tipo `A->B->C` obteremos o resultado.

3.2 Funcións sobre `nat`

Análogamente, para obter unha función definida sobre `nat` recurriremos ó esquema correspondente para `nat_rec`:

```
Coq < Definition foonat := [C:Set](nat_rec [_:nat]C).
foonat is defined
```

```
Coq < Eval Simpl in foonat.
     = foonat
     : (C:Set)C->(nat->C->C)->nat->C
```

o cual significa que para definir unha función sobre `nat` que tome valore en `C` basta fixar un elemento de `C` e dispoñer dunha colección de funcións de "`C->C` indicadas por `nat`.

3.2.1 O Producto

Por exemplo, si queremos definir a función `Producto` que faga o producto de números naturais, faremos , `C=nat->nat`, e poderemos proceder como sigue:

```
Definition Producto :=
       ( nat_rec
                  ([_:nat]nat->nat)
                  ([_:nat]0)
                  ([_:nat][f:nat->nat][x:nat](plus x (f x)))
        ).
```

[2] e si queremos estar seguros de que este programa é correcto, escribiremos a especificación:

```
Inductive producto:nat->nat->nat->Prop:=
   producto0:(m:nat)(producto 0 m 0)
  |productoS:(n,m,p:nat)(producto m n p)->(producto (S m) n
                         (plus n p )).
```

e demostraremos o seguinte:

```
Theorem test:(m,n:nat)(producto m n (Producto m n)).

Intros.
Elim m.
Simpl.
Apply producto0.
Intro.
Intro.
Simpl.
Apply productoS.
Assumption.
```

3.2.2 A función Predecesor

Outro exemplo podería ser o da función que calcula o predecesor dun número natural `Predec`, que xunto con unha especificación e unha comprobación da sua corrección móstranse a continuación.

[2] `plus` é o nome do sistema para a suma

```
Coq < Definition Predec:=(nat_rec [_:nat]nat O ([x:nat][y:nat]x)).

Inductive predecesor:nat->nat->Prop :=
     predec0:(predecesor O O)
    |predecS:(n:nat)(predecesor (S n) n).

Coq < Goal (n:nat)(predecesor n (Predec n)).

Intros.
Elim n.
Simpl.
Constructor 1.
Intros.
Simpl.
Constructor 2.
```

3.3 Recursión Primitiva

A recursión primitiva pode programarse directamente en Coq mediante a seguinte definición recursiva:

```
  Fixpoint prim_rec [C:Set;b:C;p:nat->C->C;n:nat]:C :=
        Cases n of
              O => b
         |(S x) => (p x (prim_rec C b p x))
     end.
```

cuio tipo é:

```
Coq < Check prim_rec.
prim_rec
     : (C:Set)C->(nat->C->C)->nat->C
```

pero tamén pode ser programada directamente con `nat_rec`:

```
Coq < Definition Prim_rec:=[C:Set](nat_rec [_:nat]C).
Prim_rec is defined

Coq < Check Prim_rec.
Prim_rec
     : (C:Set)
        ([_:nat]C O)
         ->((n:nat)([_:nat]C n)->([_:nat]C (S n)))->(n:nat)([_:nat]C n)

Coq < Eval Simpl in Prim_rec.
     = Prim_rec
     : (C:Set)C->(nat->C->C)->nat->C
```

e para ver que son iguais:

```
Coq < Goal (C:Set;b:C;p:nat->C->C;n:nat)(prim_rec C b p n)=(Prim_rec C b p n).
```

cuia demostración é:

```
Intros.
Elim n.
Simpl.
Trivial.
Intros.
Simpl.
Rewrite H.
Trivial.
```

4 Tipos inductivos

Referencias

[Howard] WILLIAM A HOWARD The formulae-as-type notion of construction. In H.B.Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism. *Academic Press, 1980*

[Dowek et al] GILLES DOWEK, AMY FELTY, HUGO HERBELIN, GÉRARD HUET, CHET MURTY, CATHERINE PARENT, CHRISTINE PAULIN-MOHRING E BEJAMIN WERNER The Coq proof assistant user's guide. *Technical Report 134, INRIA*,1993.

[Gordon et al] MICHAEL J. GORDON, ARTHUR J. MILNER E CHIRSTOPHER P. WADSWORTH Edinburgh LCF: A Mechanised Logic of Computation *LNCS 78, Springer* (1979)