Dos programas ás probas

G. Pérez Vega e J. L. Freire Nistal

12 de febreiro de 1999

Palabras clave: *Proba automática, Teoría de tipos, Verificación de programas, Coq, Extracción de programas, Cálculo formal*

1 Resumo

Neste traballo presentarase unha maneira de obter programas automáticamente a partires de especificacións. Esto 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 a obxectivos (*goal-directed*) e conducido por tácticas (*tactic-driven*) no estilo do LCF [Gordon et al], que permite a verificación formal de programas e, mesmo, a súa xeración automática.

O noso obxectivo consiste en amosar a potencia do extractor de programas en Coq, e utilizala para definir programas directamente dende a especificación. Esto facémolo con un exemplo que é o Algoritmo da división.

Tamén esperamos que esta reflexión sobre á ferramenta Coq sirva para despertar o interés por os probadores automáticos e mostrar que é posible, cas tecnoloxias actuais, conxugar proba e cálculo formal.

2 Introducción

A comunidade que fai Matemáticas sobor o ordeador está dividida en dous. Por unha parte están os interesados en computar e resolver e por outra os que lle interesa principalmente o tema de proba automática.

Respecto os primeiros decir que desenvolven ferramentas muy eficientes, CAS (Computer Algebra Systems), para facer as suas operacións. Entre os CAS ainos de propósito xeral como Axiom [Jenks and Sutor], Mathematica [Wolfram], Maple [Char and oth], e outros dedicados a temas específicos Macauly [Bayer and Stillman], Matlab [MATLAB]. Estos sistemas teñen moito éxito e utilízanse a miudo en matemática aplicada e inxeniería así como en temas educacionais. Os segundos tamén desenvolven ferramentas, probadores automáticos (*theorem provers*), para mecanizar as probas sobor o ordeador. Os probadores de teoremas van dende sistemas completamente automáticos como Nqthm [Boyer and Moore], ata outros principalmente interactivos como Coq [Dowek et al], HOL [Gordon and Melham], Nuprl [Constable and oth]. No medio de estes dous tipos de sistemas encóntranse os que queren encontrar unha boa relación entre automatización e interactividad como son Isabelle [Paulson] e PVS[Rushby and oth].

Na nosa opinión os probadores de teoremas teñen menos impacto cos CAS pois, mentres cos CAS sobrepasan con moito a potencia de computación human, os probadores de teoremas estan inda lexos de probar a maioría das veces o que os matemáticos poden probar fácilmente. Neste senso estase avanzando moito nos últimos tempos, proba desto podemos citar a Steve Winker que mediante o probador OTTER obtivo resultados sobor Alxebras de Robbins non demostradas anteriormente [Winker90] [Winker92].

Obviamente o ideal sería ter un marco donde a proba e a computación fose posible. Unha solución alternativa é que o sistema de proba automática sexa capaz de trasladar os algoritmos probados nun linguaxe de programación que teña un compilador. É por esto último polo que eleximos Coq para os noxos experimentos.

3 Unha breve descripción e uso de Coq

A intención desta sección non é presentar Coq con todo detalle, sinon dar unha pequena descripción para entender o código que presentamos. Pode verse unha presentación completa en [Dowek et al]. Coq é unha implementación do Cálculo de Construccións (CCI). O Cálculo de Construccións 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 teñen ese tipo. Este xeito de ver as cousas é 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.

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.

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.

Análogamente téñense `~P` para a negación, `A\/B` para a disxunción, `Ex([x:T]P)` para a cuantificación existencial, `a=b` para a igualdade, etc.

Debera notarse o carácter estrictamente constructivo das probas en Coq, cuia lóxica é intuicionista.

Os comentarios son encadrados por asteriscos (* *).

Unha construcción importante en Coq son as definicións inductivas, das cales intentaremos dar unha pequena información.

Defínese un tipo inductivo dando os constructores xunto co seu tipo. Por exemplo os naturais defínense inductivamente da forma siguiente:

```
Inductive nat : Set :=  O : nat | S : nat->nat.
```

Este tipo inductivo tén por nome **nat** e posee dous constructores **O** (a constante cero) é de tipo `nat` e **S** (función sucesor) é de tipo `nat->nat`.

Coq asocia automaticamente a cada tipo inductivo un **principio de inducción** para o tipo de proposicións e un **principio de recursión** para o tipo das especificacións.

Os principios de inducción e de recursión para **nat** son:

```
(P:nat->Prop)(P (O))->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

(P:nat->Set)(P (O))->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

A diferencia entre os dous principios reside no tipo de predicado P (`nat->Prop` ou `nat->Set`).

Tamén se pode definir predicados inductivos sobor dos obxetos que utilizamos, por exemplo a propiedade ser par:

```
Inductive par: nat -> Prop :=
   par_O: (par O)
  |par_S: (n:nat)(par n) -> (par (S(S n))).
```

Como se pode ver as dúas cláusulas son traducción en Coq dos axiomas matemáticos:

O é par.

$\forall n.$ n é par -> `(S(S n))` é par.

Debemos decir que o usuario non utiliza explicitamente os principios de inducción das definicións inductivas. A sua utilización faise a través das tácticas de eliminación, algunas das cales citamos na sección siguiente.

4 Desenvolvemento de probas en Coq

O usuario tén a posibiladade de varias sintaxis para enunciar as probas, algunha das cales son:

`Theorem<nome do teorema>: <enunciado>.`

`Lemma<nome do lema>: <enunciado>.`

`Remark<nome da nota>: <enunciado>.`

O enunciado chámase meta inicial. É decir, unha meta é unha fórmula a deducir de certas hipótesis. Estas hipótesis forman o que se chama **contexto local de hipótesis**. No caso inicial o contexto local de hipótesis esta vacio.

O desenvolvemento dunha proba faise de forma interactiva mediante ordes que se chaman tácticas. A aplicación dunha táctica a unha meta pode resolvela ou transformala en outras metas (cada unha co seu contexto local de hipótesis). O sistema dícenos en cada etapa as metas que faltan por probar. A proba acábase cando non queda ninguna meta por probar.

Existen moitas tácticas, damos un pequeno repaso a aquelas que vamos a necesitar no noso exemplo:

Intro: É unha táctica de introducción que aplicada a unha meta da forma **(x:nat)M** (ou `N->M`), introduce a hipótesis **x:nat** (repectivamente `N`) no contexto local e transforma a meta en **M**.

Elim H: permite facer probas por inducción. Esta táctica aplica o principio de eliminación asociado a definición inductiva de **H** na meta en curso. **H** debe ser un término definido inductivamente.

Exists u: Transforma a meta `<M>Ex ([x:M]N)` en `N(u/x)`. É decir en **N** reemplaza todas as ocurrencias libres de **x** por **u**, sendo **u** un termo de tipo **M**.

Simpl: Aplica todos os esquemas de recursión da meta.

Auto: Proba a meta automaticamente si é un teorema xa demostrado o un axioma.

Clear: Borra hipótesis, non necesarias.

Trivial: A meta é certa por ser unha hipótesis.

Absurd P: O argumento term é unha proposición P de tipo Prop. Xenera duas metas `~P` e `P`.

Split: Transforma a meta `N/\M` en duas metas `N` y `M`.

Left: Transforma a meta `N\/M` en `N`.

5 Proba do Algoritmo da división de naturais en Coq

Comenzamos unha sesión en Coq:

```
$ coqtop
 Welcome to Coq V6.2.2 (September 1998)
```

Cargamos librerías de Coq que necesitamos para a nosa proba. As librerías **Omega, Lt, Compare** teñén demostradas propiedades sobor os enteiros e as súas operacións, así como a relación de orde e as súas propiedades.

```
Require Export Omega.
Require Export Lt.
Require Export Compare.
```

Enunciamos o resultado que queremos probar, neste caso é o algoritmo da división.

Theorem 5.1 *Teorema (***Algoritmo da división***).*
Sean $a, b \in N$. Entonces $\exists q, r \in N$ tal que $a = q.(b+1) + r$ con $0 \leq r < (b+1)$.

Traducción en Coq:

```
Lemma div_euclid: (a,b:nat)
{q:nat*nat| a=(plus (Snd q) (mult (Fst q) (S b)))/\(lt
(Snd q) (S b))}.
```

En cada etapa da demostración en Coq faremos comentarios sobor da súa equivalencia no linguaxe matemático.

Comenzamos a proba.

```
1 subgoal

  ============================
   (a,b:nat)
    {q:(nat*nat) |
       (a=(plus (Snd q) (mult (Fst q) (S b)))/\(lt
(Snd q) (S b)))}
```

Intro a. O contexto local, inicialmente vacio, contén agora 1 hipótesis.

```
div_euclid < Intro a.
1 subgoal

  a : nat
  ============================
   (b:nat)
    {q:(nat*nat) |
     (a=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Elim a. Aplicamos o principio de inducción asociado a **nat**. Devólvenos dúas metas, unha correspondente o caso base $a = 0$ e outra a hipótesis de inducción.

```
div_euclid < Elim a.
2 subgoals

  a : nat
  ============================
   (b:nat)
    {q:(nat*nat) |
     ((0)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
     (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
```

```
   ->(b:nat)
      {q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Intro b. Introducimos no contexto local a hipótesis b.

```
div_euclid < Intro b.
2 subgoals

  a : nat
  b : nat
  ============================
   {q:(nat*nat) |
    ((0)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
     (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
   ->(b:nat)
      {q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Exists (O,O). Dámoslle como resposta $q = 0$ e $r = 0$, para resolver o caso base:

$$0 = 0 + 0(b + 1) \text{ e } 0 \leq 0 < (b + 1).$$

```
div_euclid < Exists (O,O).
2 subgoals

  a : nat
  b : nat
  ============================
   (0)=(plus (Snd ((0),(0))) (mult (Fst ((0),(0))) (S b)))
   /\(lt (Snd ((0),(0))) (S b))

subgoal 2 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
     (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
   ->(b:nat)
      {q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Simpl. Aplica os esquemas de recursión de **plus, Snd, mult, Fst**.

```
div_euclid < Simpl.
2 subgoals

  a : nat
  b : nat
  ============================
   (0)=(0)/\(lt (0) (S b))

subgoal 2 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
     (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
   ->(b:nat)
      {q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Split. Transforma a meta en dúas proposicións a demostrar. Amba las dúas son teoremas xa demostrados nas librerías cargadas o comenzo da sesión. A primeira meta é a reflexividade da igualdade de enteiros. A segunda $0 < (b+1)$ é un lema probado nas librerías, co nome de `lt_O_Sn`.

```
div_euclid < Split.
3 subgoals

  a : nat
  b : nat
  ============================
   (0)=(0)

subgoal 2 is:
 (lt (0) (S b))
subgoal 3 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
     (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
   ->(b:nat)
      {q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Reflexivity.
2 subgoals

  a : nat
  b : nat
  ============================
   (lt (0) (S b))
```

```
subgoal 2 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
     (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
   ->(b:nat)
      {q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Check lt_O_Sn.
lt_O_Sn
     : (n:nat)(lt O (S n))

div_euclid < Apply lt_O_Sn.
1 subgoal

  a : nat
  ============================
   (n:nat)
    ((b:nat)
      {q:(nat*nat) |
       (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
     ->(b:nat)
        {q:(nat*nat) |
         ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Intros n Hi b. Intros é a iteración da táctica Intro, e dámoslle nos os nomes que desexamos as hipótesis, no caso de non darlle os nomes, faino o sistema automaticamente.

```
div_euclid < Intros n Hi b.
1 subgoal

  a : nat
  n : nat
  Hi : (b:nat)
        {q:(nat*nat) |
         (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  ============================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Elim Hi with b. Aplicamos a hipótesis de inducción, para obter o cociente e o resto da división de n por $b+1$.

```
div_euclid < Elim Hi with b.
```

```
1 subgoal

  a : nat
  n : nat
  Hi : (b:nat)
        {q:(nat*nat) |
         (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  ============================
   (x:nat*nat)
    n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
     ->{q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Intros q' Hin. Introducimos a hipótesis de inducción obtida para $a = n$ e réstanos demostrar que se verifica o algoritmo para $a = n + 1$.

```
div_euclid < Intros q' Hin.
1 subgoal

  a : nat
  n : nat
  Hi : (b:nat)
        {q:(nat*nat) |
         (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  q' : nat*nat
  Hin : n=(plus (Snd q') (mult (Fst q') (S b)))/\(lt (Snd q') (S b))
  ============================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Elim Hin; Intros Hin1 Hin2. Desdoblamos as dúas proposicións da hipótesis de inducción.

```
div_euclid < Elim Hin; Intros Hin1 Hin2.
1 subgoal

  a : nat
  n : nat
  Hi : (b:nat)
        {q:(nat*nat) |
         (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  q' : nat*nat
  Hin : n=(plus (Snd q') (mult (Fst q') (S b)))/\(lt (Snd q') (S b))
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
```

```
  ===========================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Clear Hi Hin. Borramos hipótesis xa utilizadas.

```
div_euclid < Clear Hi Hin.
1 subgoal

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  ===========================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Agora réstanos probar: $\exists q, r \in N$ tal que $n+1 = q.(b+1)+r$ con $0 \leq r < (b+1)$, sabendo que temos $q', r' \in N$ verificando que: $n = q'.(b+1)+r'$ con $0 \leq r' < (b+1)$.

`Elim lt_eq_lt_dec with (S (Snd q')) (S b)`. Aplica o lema de decibilidade de dous números naturais ($r'+1$ e $b+1$) respecto a relacion binaria $<$, que está nas librerías de Coq. Obtendo tres casos:

$$(r'+1) < (b+1),\ (r'+1) = (b+1) \text{ e } (b+1) < (r'+1)$$

Pódese comprobar o enunciado do lema utilizando o comando **Check**.

```
div_euclid < Check lt_eq_lt_dec.
lt_eq_lt_dec
     : (n,m:nat){(lt n m)}+{n=m}+{(lt m n)}

div_euclid < Elim lt_eq_lt_dec with (S (Snd q')) (S b).
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  ===========================
   {(lt (S (Snd q')) (S b))}+{(S (Snd q'))=(S b)}
    ->{q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (lt (S b) (S (Snd q')))
```

```
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Separamos cada un dos casos de decibilidade, para facer a demostración por casos.

```
 div_euclid < Intro dec; Elim dec.
3 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec : {(lt (S (Snd q')) (S b))}+{(S (Snd q'))=(S b)}
  ============================
   (lt (S (Snd q')) (S b))
    ->{q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (S (Snd q'))=(S b)
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Metemos no contexto local o primeiro dos casos de decibilidade e borramos a hipótesis **dec** xa utilizada.

```
div_euclid < Intros dec1; Clear dec.
3 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec1 : (lt (S (Snd q')) (S b))
  ============================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (S (Snd q'))=(S b)
```

```
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Como estamos no caso que $r'+1 < b+1$, o cociente e o resto que verifican o lema son:

$$q = q' \text{ e } r = r' + 1.$$

```
div_euclid < Exists ((Fst q'),(S (Snd q'))).
3 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec1 : (lt (S (Snd q')) (S b))
  ============================
   (S n)
    =(plus (Snd ((Fst q'),(S (Snd q'))))
       (mult (Fst ((Fst q'),(S (Snd q')))) (S b)))
   /\(lt (Snd ((Fst q'),(S (Snd q')))) (S b))

subgoal 2 is:
 (S (Snd q'))=(S b)
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Simpl.
3 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec1 : (lt (S (Snd q')) (S b))
  ============================
```

```
   (S n)=(S (plus (Snd q') (mult (Fst q') (S b))))
   /\(lt (S (Snd q')) (S b))

subgoal 2 is:
 (S (Snd q'))=(S b)
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Debemos demostrar as dúas proposicións resultantes para o cociente e o resto dados. Resólvense mediante teoremas aritméticos demostrados nas librerías. O que atinxe o resto xa é unha hipotésis, por eso utilizamos o comando **Trivial**. En canto o cociente resulta de reescribir na meta o valor de n que temos na hipótesis **Hin1**.

```
div_euclid < Split; Trivial.
3 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec1 : (lt (S (Snd q')) (S b))
  ============================
   (S n)=(S (plus (Snd q') (mult (Fst q') (S b))))

subgoal 2 is:
 (S (Snd q'))=(S b)
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Rewrite Hin1; Trivial.
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
```

```
  dec : {(lt (S (Snd q')) (S b))}+{(S (Snd q'))=(S b)}
  ============================
   (S (Snd q'))=(S b)
    ->{q:(nat*nat) |
       ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Introducimos a hipótesis obtida no segundo caso da aplicación de decibilidade de naturais respecto a $<$, que é $r' + 1 = b + 1$.

```
div_euclid < Intros dec2; Clear dec.
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec2 : (S (Snd q'))=(S b)
  ============================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Neste caso o cociente e resto seran:

$$q = q' + 1 \text{ e } r = 0.$$

```
div_euclid < Exists ((S (Fst q')),O).
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec2 : (S (Snd q'))=(S b)
  ============================
```

```
   (S n)
    =(plus (Snd ((S (Fst q')),(0)))
       (mult (Fst ((S (Fst q')),(0))) (S b)))
   /\(lt (Snd ((S (Fst q')),(0))) (S b))

subgoal 2 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Simpl.
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec2 : (S (Snd q'))=(S b)
  ============================
   (S n)=(S (plus b (mult (Fst q') (S b))))/\(lt (0) (S b))

subgoal 2 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
     ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Desdoblamos a meta en dúas.

```
div_euclid < Split.
3 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec2 : (S (Snd q'))=(S b)
  ============================
   (S n)=(S (plus b (mult (Fst q') (S b))))

subgoal 2 is:
 (lt (0) (S b))
subgoal 3 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
```

```
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Esta meta resólvea o sistema automaticamente mediante **Omega** inda que a podemos facer paso a paso utilizando as hipótesis Hin1 e dec2.

$$\begin{gathered}(n+1) = ([b + q'(b+1)] + 1) \Rightarrow \\ n = (b + q'(b+1)) \Rightarrow \\ n = (r' + q'(b+1)).\end{gathered}$$

```
(*
Rewrite Hin1.
Apply eq_S. (* (n,m:nat)n=m->(S n)=(S m) *)
(Cut (Snd q')=b; Intros).
(Rewrite H; Auto).

Auto.

Auto.
*)

div_euclid < Omega.
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec2 : (S (Snd q'))=(S b)
  ============================
   (lt (0) (S b))

subgoal 2 is:
 (lt (S b) (S (Snd q')))
  ->{q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Apply lt_O_Sn.
1 subgoal

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  ============================
```

```
  (lt (S b) (S (Snd q')))
   ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Poñemos no contexto local de hipótesis o terceiro caso obtido cando se aplica a decibilidade de enteiros.

```
div_euclid < Intro dec3.
1 subgoal

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec3 : (lt (S b) (S (Snd q')))
  ============================
   {q:(nat*nat) |
    ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

Só nos queda probar o algoritmo no caso en que $b+1 < r'+1$. Pero eiquí temos unha contradicción ca hipótesis $r' < b+1$:

$$r' < (b+1) \Rightarrow r' \leq b.$$

e por outra parte,

$$(b+1) < (r'+1) \Rightarrow b < r'.$$

```
div_euclid < Absurd (lt b (Snd q')).
2 subgoals

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec3 : (lt (S b) (S (Snd q')))
  ============================
   ~(lt b (Snd q'))

subgoal 2 is:
 (lt b (Snd q'))
```

Este resultado aritmético podemos facelo utilizando teoremas das librerías ou directamente coa táctica **Omega**.

```
(*
Cut (le (Snd q') b).
Intros.
Apply le_not_lt. (* (n,m:nat)(le n m)->~(lt m n) *)
Trivial.
Apply lt_n_Sm_le. (* (n,m:nat)(lt n (S m))->(le n m) *)
Trivial.
*)

div_euclid < Omega.
1 subgoal

  a : nat
  n : nat
  b : nat
  q' : nat*nat
  Hin1 : n=(plus (Snd q') (mult (Fst q') (S b)))
  Hin2 : (lt (Snd q') (S b))
  dec3 : (lt (S b) (S (Snd q')))
  ============================
   (lt b (Snd q'))
```

Según o lema `lt_S_n` pra probar $b < r'$, é suficiente probar $b + 1 < r' + 1$. Esto último é a hipótesis **dec3**.

```
div_euclid < Apply lt_S_n; Trivial.
Subtree proved!
```

Para gardar a demostración do teorema utilízase o comando **Save**. Iste comando devólvenos todos os pasos da proba e ademáis autorízanos a utilizar este resultado nos siguientes teoremas a demostrar.

```
div_euclid < Save.
Intro a.
Elim a.
Intro b.
Split with (O,O).
Simpl.
Split.
Reflexivity.

Apply lt_O_Sn.

Intros n Hi b.
Elim Hi with b.
Intros q' Hin.
(Elim Hin; Intros Hin1 Hin2).
Clear Hi Hin.
```

```
Elim lt_eq_lt_dec with (S (Snd q')) (S b).
(Intro dec; Elim dec).
(Intros dec1; Clear dec).
Split with ((Fst q'),(S (Snd q'))).
Simpl.
(Split; Trivial).
(Rewrite Hin1; Trivial).

(Intros dec2; Clear dec).
Split with ((S (Fst q')),O).
Simpl.
Split.
Omega.

Apply lt_O_Sn.

Intro dec3.
Absurd (lt b (Snd q')).
Cut (le (Snd q') b).
Intros.
Apply le_not_lt.
Trivial.

Apply lt_n_Sm_le.
Trivial.

(Apply lt_S_n; Trivial).

div_euclid is defined
```

6 Extracción del algoritmo.

Unha vez rematada a proba, non só temos unha proba da correctitude do algoritmo, sinón tamén é posible extraer automaticamente unha implementación nos linguaxes **OCaml** ou **Haskell**. Para esto necesitamos primeiramente cargar a librería **Extraction**.

```
Require Extraction.
```

Unha vez cargada esta librería mediante o comando Extraction podemos extraer unha proba en Coq, a partires da cal pódese mediante os comandos **Write Caml File e Write Haskell File** extraer os ficheiros do algoritmo en OCaml ou Haskell.

```
Extraction div_euclid.
div_euclid ==>
    [a:nat]
     (nat_rec nat->(prod nat nat) [_:nat](pair nat nat
```

```
O O)
       [_:nat]
        [H:nat->(prod nat nat)]
         [b:nat]
          (sig_rec (prod nat nat) (prod nat nat)
            [x:(prod nat nat)]
             (#GENTERM (CONST #Logic#and_rec.fw) (prod
nat nat)
               (sumor_rec sumbool (prod nat nat)
                 [y:sumbool]
                  (sumbool_rec (prod nat nat)
                    (pair nat nat (fst nat nat x) (S
(snd nat nat x)))
                    (pair nat nat (S (fst nat nat x))
O) y)
                 (False_rec (prod nat nat))
                 (lt_eq_lt_dec (S (snd nat nat x)) (S
b)))) (H b)) a)
 : nat->nat->(prod nat nat)

Write Caml File "division" [div_euclid].
The axiom False_rec is translated into an exception.
Warning: The constant nat_rec is expanded.
Warning: The constant fst is expanded.
Warning: The constant snd is expanded.
Warning: The constant and_rec is expanded.
Warning: The constant sig_rec is expanded.
Warning: The constant sumbool_rec is expanded.
Warning: The constant sumor_rec is expanded.

Write Haskell File "division" [div_euclid].
The axiom False_rec is translated into an exception.
```

Os ficheiros extraidos son:
En **OCaml**.

```
type nat =
    O
  | S of nat
;;

type ('a, 'b) prod =
    Pair of 'a * 'b
;;

type sumbool =
    Left_ren
  | Right_ren
```

```
;;

type 'a sumor =
    Inleft of 'a
  | Inright
;;

let lt_eq_lt_dec n =
  let rec f1 = function
    O -> (fun m ->
      let rec f3 = function
        O -> Inleft Right_ren
      | S n2 -> Inleft Left_ren
      in f3 m)
  | S n1 -> (fun m ->
      let rec f2 = function
        O -> Inright
      | S n4 ->
          (match f1 n1 n4 with
             Inleft x ->
               (match x with
                  Left_ren -> Inleft Left_ren
                | Right_ren -> Inleft Right_ren)
           | Inright -> Inright)
      in f2 m)
  in f1 n
;;

let div_euclid a =
  let rec f1 = function
    O -> (fun b -> Pair (O, O))
  | S n0 -> (fun b ->
      match lt_eq_lt_dec (S (match f1 n0 b with
                               Pair (x, y) -> y)) (S b) with
        Inleft x ->
          (match x with
             Left_ren ->
               Pair ((match f1 n0 b with
                        Pair (x0, y) -> x0),
                     (S (match f1 n0 b with
                           Pair (x0, y) -> y)))
           | Right_ren ->
               Pair ((S (match f1 n0 b with
                           Pair (x0, y) -> x0)), O))
      | Inright -> failwith "False_rec")
  in f1 a
;;
```

En **Haskell**.

```
data Nat = O
         | S Nat

nat_rec f f0 =
  let { f1 n =
    case n of
      O -> f
      S n0 -> f0 n0 (f1 n0) }
  in f1

data Prod a b = Pair a b

fst p =
  case p of
    Pair x y -> x

snd p =
  case p of
    Pair x y -> y

and_rec f =
  f

sig_rec f s =
  f s

data Sumbool = Left_ren
             | Right_ren

sumbool_rec f f0 s =
  case s of
    Left_ren -> f
    Right_ren -> f0

data Sumor a = Inleft a
             | Inright

sumor_rec f f0 s =
  case s of
    Inleft x -> f x
    Inright -> f0

false_rec =
  error "False_rec"

lt_eq_lt_dec n =
```

```
  nat_rec
    (\m -> nat_rec (Inleft Right_ren) (\n0 h -> Inleft Left_ren) m)
    (\n0 h m -> nat_rec Inright
                 (\q h' -> sumor_rec
                             (\y -> sumbool_rec (Inleft Left_ren)
                                      (Inleft Right_ren) y) Inright
                             (h q)) m) n

div_euclid a =
  nat_rec (\b -> Pair O O)
    (\n hi b -> sig_rec
                 (\q' -> and_rec
                           (sumor_rec
                             (\dec -> sumbool_rec
                                        (Pair (fst q') (S (snd q')))
                                        (Pair (S (fst q')) O) dec)
                             false_rec
                             (lt_eq_lt_dec (S (snd q')) (S b))))
                 (hi b)) a
```

7 Conclusión e traballo futuro.

Unha vez feito o desenvolvemento da proba tense a sensación de que probar require moito más esforzo que programar. Inda que esto non é unha sorpresa, indica que a perspectiva de facer un sistema CAS completamente certificado non é realista polo momento, pero cremos que hay moito que gañar no desenvolvemento de algoritmos de álxebra computacional en sistemas donde se poda razoar acerca deles.

Sería interesante realizar a mesma proba noutro sistema, co cal daríanos un punto vista máis exacto acerca da tecnoloxia a utilizar para este algoritmo en particular. Tamén é importante complementar este intento con outros algoritmos máis complicados, por exemplo o **Algoritmo de Buchberger** para calcular unha **Base de Groebner** dun conxunto de polinomios. Iste algoritmo, no cal estamos traballando tén o problema engadido de ter que construir os polinomios así como as operacións sobre eles e as súas propiedades.

Para rematar decir que a demostración anteriormente feita do **Algoritmo da división** pódese acortar e **limpar** bastante; non o fixemos pous deste modo sería máis dificil o seguimento da proba.

Referencias

[Bayer and Stillman] D. Bayer and M. Stillman A system for computation in algebraic geometry and commutative algebra. Available at http://www.math.columbia.edu/- bayer/macauly.html, 1994.

[Boyer and Moore] R.S. Boyer and J.S. Moore A computational Logic handbook. *Academic Press, 1988* .

[Char and oth] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, S. Watt A Tutorial Introduction to Maple V. *Springer-Verlag, 1992* .

[Constable and oth] R. Constable, S. Allen, W. Cleaveland, H. Bromley, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden; J. Sasaki, and S. Smith Implementing Mathematics with the Nuprl. *Prentice Hall, 1986* .

[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).

[Gordon and Melham] M.J.C. Gordon and T.F. Melham Introduction to HOL. *Cambridge, 1993* .

[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* .

[Jenks and Sutor] Richard D. Jenks e Robert S. Tutor AXIOM: the scientific computation system. *Springer-Verlag, 1992* .

[MATLAB] MATLAB matrix software, The Mth Works, Inc., 158 Woodland St., Sherborn, MA01770.

[Paulson] L.C. Paulson Isabelle: A generic theorem prover. *LNCS 828, Springer* (1994).

[Rushby and oth] J. Rushby, N. Shankar, and M. Srivas PVS: combining specification, proof checking, and model checking. *LNCS 1102, Springer* (1996).

[Winker90] S. Winker Robbins Algebra: Conditions That Make a Near-Boolean Algebras. *Automated Reasoning 6,no.4:465-489* (1990).

[Winker92] S. Winker Absorption and Idempotency Criteria for a Problem in Near-Boolean Algebras. *J. Algebra 153, no.2:414-423* (1992).

[Wolfram] S. Wolfram Mathematica: A system for doing mathematics by computer. *Addison Wesley, 1991* .