El asistente de pruebas Coq.
Lógica Computacional 2017-2

Lourdes del Carmen González Huesca
Roberto Monroy Argumedo
Fernando A. Galicia Mendoza

Facultad de Ciencias,UNAM

Miércoles,2 de mayo del 2017

En búsqueda de la formalización de las matemáticas

Recordemos que David Hilbert propuso un programa el cual contuviera toda la teoría matemática y mediante un procedimiento finito demostrara teoremas de manera automática. Tiempo después Gödel contradijo este hecho con el segundo de sus teoremas.
Sin embargo, como mucha teoría matemática fue analizada a profundidad, con lo que matemáticos y computologos de la historia descubrieron dos teorías/implementaciones: demostradores automáticos y asistentes de prueba.

Principio fundamental de los asistentes

El trabajo en base de un computologo es implementar teoría para la resolución de problemas con la certeza de que una demostración garantiza la solución de ese problema (o su imposible resolución, de igual forma debe ser demostrada).

Principio fundamental de los asistentes

El trabajo en base de un computologo es implementar teoría para la resolución de problemas con la certeza de que una demostración garantiza la solución de ese problema (o su imposible resolución, de igual forma debe ser demostrada).
Cuando uno demuestra está en la línea delgada de cometer un error humano, esto se debe a que tenemos energía limitada, las cuentas pueden ser tan largas o díficiles que necesitamos un equipo de cómputo que nos haga tales cuentas o bien, la demostración es tan monstruosamente larga que existe la posibilidad que olvidemos los pasos por los que hemos venido o simplemente cometer un error de dedo.

Principio fundamental de los asistentes

El trabajo en base de un computologo es implementar teoría para la resolución de problemas con la certeza de que una demostración garantiza la solución de ese problema (o su imposible resolución, de igual forma debe ser demostrada).
Cuando uno demuestra está en la línea delgada de cometer un error humano, esto se debe a que tenemos energía limitada, las cuentas pueden ser tan largas o díficiles que necesitamos un equipo de cómputo que nos haga tales cuentas o bien, la demostración es tan monstruosamente larga que existe la posibilidad que olvidemos los pasos por los que hemos venido o simplemente cometer un error de dedo.
Los asistentes de pruebas fueron ideados con dos principios básicos: la computadora no se equivoca en cuentas (asumiendo que el programa utilizado no esté corrupto) y la lógica nos brinda el lenguaje formal con el cual la computadora nos logra entender.

Historia de Coq

"Coq es el resultado de aproximadamente 30 años de investigación. Iniciado en 1984 de una implementación del cálculo de construcciones en INRIA (Institut National de Recherche en Informatique et en Automatique, por sus siglas en francés) por Thierry Coquand y Gérard Huet. En 1991, Christine Paulin lo extendió al cálculo de construcciones inductivas." - Fragmento obtenido de `https://coq.inria.fr/about-coq`.

Usos y no usos del asistente

Usos del asistente:

- Verificar de manera formal teoría matemática o implementaciones de la solución de un problema.
- Análisis profundo sobre los teoremas demostrados a mano.

NO usos del asistente:

- Verificación de modelos de sistemas, por ejemplo el modelo que representa el *three hand shake* es mejor demostrar sus propiedades utilizando un demostrador automático, siendo mas preciso un *model checker*.
- Algunos consideran que no debe ser utilizado para ingeniería del software, debido a su proceso lento de obtención de implementaciones.

Ambiente de Coq

Existen dos posibles editores para trabajar con Coq: CoqIDE y Proof General. No hay alguno mejor, es cuestión de gustos.
Recomendación del ayudante: Para los que llevan trabajando con Emacs es recomendable trabajar con Proof General, ya que tienen un mejor manejo de comandos y harán más fácil su trabajo, desde mi punto de vista.
Veamos como está el ambiente.

`Gallina` ... puras aves

`Gallina` es una especificación formal del lenguaje que reconoce el núcleo del asistente y está básicamente dividido en tres grupos esenciales: vernáculos, tácticas y sentencias.
Los vernáculos inician en mayúsculas y son palabras reservadas que le indican al asistente: si se definirá un objeto de análisis (definiciones matemáticas, estructuras de datos, secciones, clases, etc.)
Las tácticas son minúsculas y son para la parte de demostración.
Por último las sentencias son las palabras reservadas para la parte de programación, por ejemplo `match` es para realizar caza de patrones.

Mantras

Hay dos cosas que tener bien claras al momento de trabajar con `Coq`:

- Todo es un tipo: La parte de programación funcional no deberá haber problema alguno, sin embargo, en lo que nos adentramos observaremos que las definiciones matemáticas y demostraciones son obtenidas gracias a la correspondencia Curry-Howard. También veremos que los constructores son realmente tipos función.
- Toda función debe ser total: No puede haber funciones parciales, es decir, todo elemento del dominio deberá corresponderle al menos un elemento de la imagen. Funciones como `head` podremos dar su equivalente o dar hipótesis necesarias para que resulte a la función implementada en `Haskell`.

Definiciones y funciones no recursivas

El vernáculo `Definition` por el momento nos servirá para dos fines: definir tipos no recursivos y funciones no recursivas.
Su sintaxis es:

```
Definition nombre (x1:parametro) ... (xn:parametro) : tipo :=
Cuerpo_definicion_funcion.
```

Por ejemplo el tipo par de naturales y booleanos se implementa de la siguiente forma:

```
Definition parNB := prod nat bool.
```

O la función `iZ` (es cero) se define:

```
Definition iZ (n:nat) : bool :=
match n with
| 0 ⇒ true
| _ ⇒ false
end.
```

Definiciones recursivas y relaciones

Coq nos permite definir tres tipos esenciales de definiciones recursivas utilizando el vernáculo `Inductive`: tipos recursivos, estructuras recursivas y relaciones.

```
Inductive nats : Type :=
| C : nats
| Su : nats → nats.
Inductive lista (A:Type) : Type :=
| Nil : lista A
| Cons : lista A → A → lista A.
Inductive MeI : nat → nat → Prop :=
| n_n (n:nat) : MeI n n
| n_S_m (n m:nat) : MeI n m → MeI n (S m).
```

Un lenguaje muy poderoso

Como acaban de observar `Coq` como lenguaje de programación es bastante poderoso, nos permite programación funcional y lógica, las dos primeras son fácilmente implementables en `Haskell` pero la tercera no podría implementarse, sin embargo, en `Prolog` si es posible.
Lo que podría causar conflicto es el tipo `Prop`, este tipo es un monstruo en el sentido que es inimaginablemente grande, ya que, este tipo contiene las proposiciones que se te ocurran. Por ejemplo la expresión `1 + 2 = 3` es una proposición en la teoría de números y pertenece a el tipo `Prop`.

Funciones recursivas primitivas

Para definir funciones recursivas **primitivas** se utiliza el vernáculo `Fixpoint` y su sintaxis es la misma que `Definition`.
Difieren en ejecución, ya que, `Fixpoint` sabrá que en algún momento puede mandarse a llamar la función, a diferencia que `Definition` cuando mande a llamar la función dirá que esta no ha sido definida.
La recursión primitiva es sobre sintaxis, es decir, cuando se llama a la función debe ser sobre subtérminos del término original, es decir, debe tener menos constructores que la entrada dada.
Por ejemplo: `n` es subtérmino de `S n`, `i` es subtérmino de `Nodo x i d`, pero `Neg phi` **no** es subtérmino de `phi` (instrucción esencial para la forma normal negativa).

¿Limitante? Sí, pero con justificación

"Todo poder conlleva una gran responsabilidad."

Coq no es la excepción a esta frase y es que hemos visto sus bondades, pero veamos sus restricciones, **no** (y ténganlo bien en cuenta): En este curso no manejaremos la recursión general en Coq, a pesar de su "inocencia" son resultados fuertes de la teoría de conjuntos y sabiamente Coq nos restringe este tipo de definiciones.
Ya que al permitir este tipo de definiciones, Coq debería permitir demostraciones infinitas. . . por la lógica que tiene detrás.
Sin embargo, es posible definir funciones recursivas generales pero tienen que ver con buenos órdenes y construcción de relaciones que son temas de cursos avanzados de programación.

Ver para creer

Ejercicios:

- Empecemos con un caso sencillo: el tipo lista y las funciones head y map.
- Implementemos los binarios positivos y las funciones: sucesor, predecesor positivo y suma.
- Definamos los árboles binarios y las funciones: esVacio, mínimo y máximo.
- Definamos la relación ser par y la relación ser impar de números naturales.