Verificación Formal

PCIC IIMAS UNAM

Sesión 1: Introducción

Favio Ezequiel Miranda Perea

Departamento de Matemáticas
Facultad de Ciencias UNAM

26 de octubre de 2020

Introducción

- Objetivo principal:

  Proporcionar evidencia de que un programa o sistema tiene o cumple el comportamiento deseado.

- Nociones centrales:
  - Modelo
  - Especificación

- Estos conceptos y su realización permiten un tratamiento matemático del objeto de interés

Introducción

- El objetivo principal puede separse en dos subproblemas:
  - Garantia de comportamiento:
    - ¿Cómo asegurar o verificar que un modelo cumple un comportamiento dado ?
  - Modelo vs. implementación:
    - ¿Cómo obtener, a partir del modelo, una implementación que cumple el comportamiento ?
    - ¿Cómo asegurar que una implementación (un código de programa) dada tiene el mismo comportamiento que el modelo ?
- Buscamos una herramienta que proporcione soporte para ambos subproblemas: modela, prueba y extrae código a un lenguaje de programación.

Computer Programs for Checking Mathematical Proofs

John McCarthy 1962

Proof-checking by computer may be as important as proof generation. It is part of the definition of formal system that proofs be machine checkable

⋮*For example, instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties*

Especificación y demostración de propiedades

- 3 niveles de soporte para la verificación formal (J. Rushby)
  - Herramientas que proporcionan un ambiente formal únicamente: las pruebas son a mano, expresadas en lenguaje natural y se validan mediante consenso de la comunidad experta.
  - Herramientas que proporcionan un sistema formal para la formulación precisa del razonamiento: las pruebas son a mano pero expresadas en un lenguaje riguroso.
  - Herramientas que proporcionan un sistema formal y soporte computacional para la formulación precisa del razonamiento: tanto la definición del modelo como las pruebas de sus propiedades se desarrollan en el mismo marco y su verificación sistemática también.
- Necesitamos una herramienta que pertenezca al último nivel.

Pruebas asistidas por computadora

- Una prueba formal consiste de razonamientos (reglas de inferencia) y cómputos (reducciones)
- La computadora puede asistir en ambas partes:
  - Cómputos:
    - ★ Numéricos (calculadoras)
    - ★ Simbólicos (MAPLE, MATHEMATICA)
  - Razonamiento:
    - ★ Automático: demostradores automáticos de teoremas, verificadores de modelos, solucionadores SAT/SMT
    - ★ Interactivo: asistentes de prueba

Sistemas formales de prueba

- Por sistema formal entenderemos un sistema lógico deductivo (en nuestro caso el cálculo de secuentes)
- Los sistemas formales en este sentido se clasifican mediante la relación entre dos factores a tomar en cuenta:
  - La expresividad de la lógica
  - La automatización del razonamiento
- Tenemos dos clases de importancia: los demostradores automáticos y los asistentes de prueba

Asistentes vs. demostradores automáticos

- Demostrador automático de teoremas: sistema computacional que permite al usuario verificar la validez de teoremas generándolos automáticamente, mediante implementaciones de algoritmos adecuados como la resolución binaria.

- Asistente de prueba: sistema computacional que permiten al usuario generar pruebas de manera interactiva, de manera que el usuario es quien tiene el control del proceso de derivación.

¿Qué es un asistente de prueba ?

AP

- Un sistema computacional que permite al usuario generar pruebas de manera interactiva, de manera que es él quien tiene el control del proceso de inferencia.
- Mediante el uso de tácticas dictadas por el usuario, el asistente genera un código de prueba que corresponde a una prueba matemática estandar
- Un AP asiste al humano de diversas maneras:
  - Formalización de teorías o especificaciones: definiciones, axiomas, etc..
  - Verificando y generando pruebas en un lenguaje lógico como la deducción natural, mediante la automatización de heurísticas (tácticas) de prueba.
  - Administración: proporcionando herramientas (editores, bibliotecas, documentación, extracción de programas).

Asistentes de prueba

- Preferencia: expresividad lógica (lógica de orden superior)
- Ventajas: se puede expresar una gran clase de conceptos, propiedades y pruebas que se requieren para una verificación formal completa de sistemas complejos. El razonamiento aquí es similar al razonamiento matemático estandar.
- Desventajas: la indecidibilidad de la lógica subyacente, lo cual implica la intervención frecuente del usuario.
- Sistemas: Isabelle/Hol, Hol Light, Matita, Lean, Coq,...

Coq según Coq

http://coq.inria.fr

- Coq es el sistema ganador del premio ACM SIGPLAN 2013 para software de lenguajes de programación.

- De la página de Coq: *Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the formalization of programming languages semantics (e.g. the CompCert compiler certification project or Java Card EAL7 certification in industrial context), the formalization of mathematics (e.g. the full formalization of the 4 color theorem or constructive mathematics at Nijmegen) and teaching.*

El asistente de pruebas Coq

http://coq.inria.fr

- ¿ Qué es Coq ?
  - Una lógica de orden superior (el cálculo de construcciones inductivas) implementada.
  - Un lenguaje de programación funcional sofisticado
  - Una herramienta robusta de desarrollo de pruebas
- ¿ Para qué se usa Coq ?
  - Para desarrollar pruebas matemáticas (Teorema de los 4 colores)
  - Para desarrollar software certificado (Compilador CompCert de C)
  - En general para desarrollar de manera interactiva pruebas formales.

Coq

Generalidades

- Coq corresponde a la clase de métodos formales pesados, de propóstio general, muy poderoso y expresivo.
- Sus principales usos son:
  - Verificación y certificación formal de programas
  - Formalización de las matemáticas.
- En su forma actual no es una herramienta que se pueda integrar en un proceso regular de desarrollo de software. Es artesanal.
- Sin embargo se usa de manera exitosa en la industria.
- Pertenece a la familia de herramientas recomendadas por la norma ISO/IEC 15408 para seguridad en cómputo. (*Common Criteria for Information Technology Security Evaluation*)

Especificación/Verificación

Mini ejemplo

- Especificación: la función *sumh* recibe un natural n y devuelve la suma de todos los naturales hasta n. Es decir,

$$sumh\ n = 0 + 1 + 2 + \ldots + n$$

- Implementación: se define la función *sumn* recursivamente como sigue:

```
sumh 0 = 0
sumh (S n)  = sumh n + S n
```

- ¿ La implementación cumple la especificación ?

$$\forall n \in \mathbb{N}\,(\texttt{sumh}\, n = \sum_{i=0}^{n} i)$$

Especificación/Verificación

Mini ejemplo

- Especificación: la función *sumh* recibe un natural n y devuelve la suma de todos los naturales hasta n. Es decir,

$$sumh\ n = 0 + 1 + 2 + \ldots + n$$

- Implementación: y si hubieramos definido

```
sumh 0 = 666
sumh (S n)  = 1 + 2n + n^2
```

- ¿ La implementación cumple la especificación ?

$$\forall n \in \mathbb{N}\,(\texttt{sumh}\, n = \sum_{i=0}^{n} i)$$

Especificación/Verificación

Mini ejemplo

- Especificación: la función *sumh* recibe un natural n y devuelve la suma de todos los naturales hasta n. Es decir,

$$sumh\ n = 0 + 1 + 2 + \ldots + n$$

- Y si hubieramos definido

```
sumh n  = n(n+1)/2
```

- ¿ La implementación cumple la especificación ?

$$\forall n \in \mathbb{N}\ (\texttt{sumh}\ n = \sum_{i=0}^{n} i)$$

¿ Cómo usar Coq ?

- Idea fundamental: Verificación $\approx$ Teorema

  Teorema. El programa p es correcto

- En el mismo ambiente se define el programa y la prueba.
- Etapas del proceso de certificación:
  - Parte funcional: datos y sus operaciones (programa)
  - Parte lógica: propiedades y sus pruebas (especificación)
- Caso particular: un programa para ordenar listas.
- Datos:

```
Inductive listZ: Type := nilZ |
                         consZ (hd:Z) (tl: listZ).

Notation ‘‘hd::tl’’ := consZ hd tl
```

Estudio de un caso

Ordenamiento por inserción

- Operaciones:

```
Fixpoint insert (x : Z) (l: list Z) :=
 match l with
   | nilZ => x::nilZ
   | hd::tl => if x <= hd then x::l
                  else hd::insert x tl
end.

Fixpoint sort l :=
 match l with
   | nilZ => nilZ
   | hd::tl => insert hd (sort l)
end.
```

Estudio de un caso

Ordenamiento por inserción

- Propiedades: ser una lista ordenada

```
Inductive sorted : list Z -> Prop :=
  | sorted0 : sorted nil
  | sorted1 : forall z:Z, sorted (z :: nil)
  | sorted2 :
      forall (z1 z2:Z) (l:list Z),
        z1 <= z2 ->
        sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

Estudio de un caso

Ordenamiento por inserción

- Pruebas: `sort` produce una lista ordenada, insertar un elemento a una lista ordenada genera una lista ordenada

```
Theorem sort_correct : forall (l:list z), sorted (sort l).

Theorem sort_ins : forall (z:Z) (l:list Z),
                        sorted l -> sorted (insert z l).

Theorem sorted_inv : forall (z:Z) (l:list Z),
                          sorted (z :: l) -> sorted l.
```

Equivalencia de listas

```
Fixpoint nb_occ (z:Z) (l:list Z) : nat :=
  match l with
   | nil => 0%nat
   | x::xs => match Z_eq_dec z x with
                | right _ => (nb_occ z xs)
                | left _ => S (nb_occ z xs)
              end
end.
```

```
Definition equiv (l l' : list Z) :=
      forall (z:Z), ((nb_occ z l) = (nb_occ z l')).

Lemma equiv_ref : forall (l : list Z), equiv l l.
```

Equivalencia de listas

```
Lemma equiv_sym : forall (l l' : list Z),
                  equiv l l' -> equiv l' l.

Lemma equiv_trans : forall (l1 l2 l3 : list Z),
                    equiv l1 l2 ->
                     equiv l2 l3 ->
                      equiv l1 l3.

Lemma equiv_cons : forall (l1 l2 : list Z) (z:Z),
                    equiv l1 l2 ->
                       equiv (z::l1) (z::l2).

Lemma equiv_perm : forall (l1 l2 : list Z) (z1 z2 : Z),
                   equiv l1 l2 ->
                    equiv (z1::z2::l1) (z2::z1::l2).
```

Extracción de programas certificados

- Prueba

```
Theorem sort :
  forall l:list Z, exists (l' : list Z),
     equiv l l' /\ sorted l'.
 Proof.
  induction l as [| a l IHl].
  exists (nil (A:=Z)); split; auto with sort.
  case IHl; intros l' [H0 H1].
  exists (aux a l'); split.
  apply equiv_trans with (a :: l'); auto with sort.
  apply aux_equiv.
  apply aux_sorted; auto.
 Qed.
```

- Extracción de un programa a partir de la prueba:

```
Extraction "insert-sort" aux sort.
```

Aplanamiento de un árbol binario

Casos introductorios

- Sea t un árbol binario t
- Sea `flat` una función que devuelve una lista con los elementos de un árbol.
- Verificar la propiedad de invarianza de `flat` respecto a los elementos de t.
- x pertenece a t si y sólo si x pertenece a `flat` t.

Números naturales por paridad

Casos introductorios

- Un natural `nat` es el cero o bien el sucesor de un natural

$$n ::= 0 \mid s\ n$$

- Un natural por paridad `pnat` es el cero o bien el doble de un natural más uno o bien el doble de un natural más dos. Es decir, el cero, los impares y los pares no cero.

$$p ::= 0 \mid d\ p \mid u\ p$$

- Mostrar la equivalencia entre `nat` y `pnat`
- Verificar que las funciones aritméticas $+, *$ coinciden.
- Definir relaciones de orden y verificar que coinciden.

Propiedades de la estructura de datos lista

Casos introductorios

Las funciones `take` y `drop` se especifican como sigue:

- `take n xs` devuelve la lista con los primeros *n* elementos de la lista *xs*.
- `drop n xs` devuelve la lista resultante de eliminar los primeros *n* elementos de *xs*.
- Demuestre las siguientes propiedades:
  1. `xs = take n xs ++ drop n xs`
  2. `take m . drop n = drop n . take (m+n)`
  3. `take m . take n = take (min m n)`
  4. `drop m . drop n = drop (m+n)`

Optimización de expresiones aritméticas

Casos introductorios

$$e := n \mid x \mid e + e$$

- Definir una función `simpl` que simplifique las sumas de números, al nivel sintáctico.
- Por ejemplo `simp` (x+(5+2)) = x+7
- Verificar la corrección de `simp` respecto al intérprete.

$$eval\ e\ s = eval\ (simpl\ e)\ s$$

- Optimizar ahora las sumas con cero y volver a verificar.

Equivalencia en definiciones de latiz

Casos introductorios

- Def 1. Una latiz es una estructura algebraica con dos operaciones binarias $\sqcap, \sqcup$ que cumplen ciertas propiedades.
- Def 2. Una latiz es un orden parcial $\leq$ donde cualesquiera dos elementos tienen supremo e ínfimo.
- Verificar que ambas definiciones son equivalentes.