Taller de Especificación, Construcción y Verificación Formales de Programas

Propuesta y Experiencias

Carlos Daniel Luna
Instituto de Computación, Facultad de Ingeniería, Universidad de la República
Julio Herrera y Reissig 565 - Piso 5, (cp:11300), Montevideo, Uruguay
TE: (+598) (2) 7114244 (int. 115). Fax: (+598) (2) 7110469
cluna@fing.edu.uy

Resumen

En este trabajo presentamos una propuesta para apoyar la enseñanza de métodos formales en una currícula de grado usando el asistente de pruebas Coq y conceptos del área de Teoría de Tipos. Proponemos un taller de especificación, construcción y verificación de sistemas en los paradigmas de programación funcional e imperativo, que también abarca el análisis de sistemas críticos: sistemas reactivos y de tiempo real. Describimos algunas experiencias en el desarrollo del taller y planteamos cambios y extensiones.

Palabras claves: Enseñanza de la Programación, Teoría de Tipos, Coq, Programación Funcional, Programación Imperativa, Sistemas Reactivos y de Tiempo Real.

Dirigido al "XII Ateneo de Profesores Universitarios de Computación"

1. INTRODUCCIÓN

Los profesionales de la computación deben estar capacitados para estudiar los fundamentos de su disciplina. El núcleo central de las Ciencias de la Computación está constituido en buena parte por la matemática discreta y la lógica matemática. En consecuencia, un especialista en computación debe estar en condiciones de usar las herramientas básicas y las técnicas de dichas áreas. Esto le permitirá una adecuación rápida y eficaz a los acelerados cambios tecnológicos, que son una constante en la disciplina.

En este trabajo proponemos un taller que tiene como meta la adquisición de destreza en el uso de herramientas de razonamiento fundamentales: la inducción matemática y la deducción lógica aplicadas a la construcción y verificación de programas. Los objetivos centrales que se persiguen pueden resumirse en la frase "programar rigurosamente sobre la base de argumentos matemáticos". Esto es, fortalecer la noción de que junto con la construcción de los algoritmos existe la obligación de la verificación rigurosa (formal) de su corrección y que los programas son objetos matemáticos plausibles de ser tratados con argumentos lógico-matemáticos [8].

Para lograr estos objetivos presentamos un taller para apoyar la enseñanza de métodos formales en una currícula de grado, usando el asistente de pruebas *Coq* [1] y conceptos del área de *Teoría de Tipos*. El taller abarca la especificación, construcción y verificación de sistemas en los paradigmas de programación funcional e imperativo y, el análisis de sistemas críticos: programas reactivos y de tiempo real.

La estructura del artículo es como sigue. En la sección 2 describimos sucintamente las metodologías más usadas para desarrollar programas correctos y verificar corrección. En la sección 3 destacamos las principales características del asistente de pruebas *Coq* y en la sección 4 desarrollamos algunas de estas características para ilustrar la utilidad de *Coq* como asistente para programadores. En la sección 5 presentamos nuestra propuesta, un taller de construcción de programas certificados usando *Coq*. En la sección 6 exhibimos algunas experiencias en el desarrollo del taller y finalmente, en la sección 7, incluimos las conclusiones de este artículo. Una versión preliminar de este trabajo fue presentada en la *Conferencia Latinoamericana de Informática: CLEI'2002 – CIESC'2002* [18].

2. DEMOSTRACIÓN Y VERIFICACIÓN DE CORRECCIÓN

Es indiscutible hoy la influencia que tiene en la industria y en casi todos los ámbitos el uso del software. La cantidad de aplicaciones reales y potenciales de la computación ha alcanzado cotas inimaginables apenas veinte años atrás. A pesar de su uso extensivo, uno de los costos más alto no se da en la producción del software, sino en la corrección de errores que son detectados posteriormente al desarrollo del sistema. En la actualidad, el método más usado para validar software es el "*testing*", que consiste en la simulación sobre casos de prueba representativos. No obstante, este método no garantiza la corrección del software analizado, por ser incompleto en la mayoría de los casos [21]. En las aplicaciones críticas, que tratan con vidas humanas y/o grandes inversiones económicas, la *certeza de corrección* es, en general, un criterio indispensable. De un software correcto se espera que resuelva un problema determinado por una *especificación* y que exista una justificación formal –matemática– de que el programa la satisface.

En los últimos años un gran esfuerzo de investigación se ha invertido en el desarrollo de métodos y herramientas para la especificación y el análisis de la corrección de sistemas. Sin embargo no hay un formalismo, una metodología o una herramienta claramente preferibles a otras en todas circunstancias. Para el análisis de la corrección formal de sistemas se destacan dos importantes enfoques:

- ***Verificación de corrección***. En este enfoque un sistema es considerado correcto cuando se prueba que *toda* ejecución posible satisface la especificación. Existen algunas técnicas bien conocidas que permiten recorrer, en ciertos casos, de manera exhaustiva el espacio de ejecuciones posibles y herramientas que las implementan.
- ***Demostración de corrección***. En este caso se construye o deriva una prueba matemática de que el sistema satisface su especificación. Aquí las herramientas asisten al programador en el proceso de construcción de la prueba. Algunas de estas herramientas están basadas en teorías constructivas de tipos [2, 3, 4], las cuales han sido formuladas como fundamento de la Matemática Constructiva. Ejemplos de estos sistemas son ALF [20], Coq [1] y LEGO [19]. Una de las principales características de los mismos es el carácter unificador de la teoría que implementan, en la cual pueden ser expresados programas, teoremas y pruebas de éstos. Otro punto destacable es que el usuario es guiado en forma interactiva por el sistema en el proceso de construcción de un programa o una prueba, siendo verificada inmediatamente la validez de cada paso del desarrollo. El principal objetivo de estos sistemas es convertirse en sofisticadas herramientas que asistan en la tarea del desarrollo incremental de programas correctos. Sin embargo, el marco conceptual necesario para desarrollar software verificado es de una muy alta complejidad y requiere cubrir muchos aspectos que en realidad escapan a la construcción de un asistente de pruebas. Estos sistemas disponen de un lenguaje de especificación de orden superior, permiten hacer pruebas en lógica de alto orden y proveen definiciones de tipos inductivos y co-inductivos.

En este artículo usaremos los términos verificación y demostración de corrección indistintamente de aquí en adelante, salvo expresa acotación, refiriéndonos conceptualmente siempre a este último tipo de análisis de corrección.

3. ACERCA DE COQ

El asistente de pruebas *Coq* es una implementación del *cálculo de construcciones inductivas*, una lógica intuicionista de alto orden con tipos dependientes y tipos inductivos como objetos primitivos [3, 25]. El usuario introduce definiciones y hace demostraciones en un estilo de *deducción natural*, las cuales son chequeadas mecánicamente por el sistema.

Dicho formalismo permite especificar y probar en lógica intuicionista de alto orden. Esta lógica asocia una interpretación computacional a las pruebas, la noción de veracidad de una proposición corresponde a la existencia de una prueba. Curry y Howard demostraron que los siguientes juicios son equivalentes:

- "*t* es una demostración de la proposición *A*".
- "el término t tiene tipo A".
- "t es un programa de la especificación A".

Esto es, probar una proposición *A* es equivalente a construir un término de tipo *A*. Esta equivalencia es conocida como isomorfismo de Curry-Howard [16] y está basada en que las pruebas en deducción natural pueden ser representadas como términos de un cálculo lambda tipado [16], o que es lo mismo, de un lenguaje de programación funcional (por ejemplo, *Caml* [26]). Consecuentemente *Coq* puede ser considerado, rigurosamente hablando, un chequeador de tipos ("type-checker").

A los efectos de diferenciar objetos computacionales de información lógica, distinguimos dos clases de tipos importantes en *Coq*: *Prop* para los tipos que contienen términos lógicos (las proposiciones) y *Set* que agrupa a los tipos que contienen información computacional (los conjuntos). Por ejemplo, los números naturales se definen en *Set* y las relaciones $\leq$ y la conjunción $\wedge$ en *Prop*. *Set* y *Prop* pertenecen a *Type*, que es en realidad una familia infinita de tipos notada de esta forma. Un procedimiento de *extracción de programas* puede ser usado para remover las partes lógicas de los términos, manteniendo sólo las partes de información computacional [23, 24].

Además de los habituales tipos inductivos (o sea, conjuntos definidos inductivamente, como por ejemplo los números naturales o las listas finitas), *Coq* permite también la definición de tipos *co-inductivos*. Estos son tipos recursivos que pueden contener objetos infinitos, no bien fundados. Un ejemplo de tipos co-inductivos es el de las secuencias infinitas, usualmente llamadas *streams*, de elementos de un tipo dado.

En el proceso de prueba de un teorema, *Coq* entra en un ciclo interactivo donde el usuario completa la demostración usando *tácticas*, las cuales implementan reglas de inferencia o de tipado (esquemas de prueba). El conjunto de tácticas puede ser incrementado por el usuario, a partir de un lenguaje diseñado para tal fin.

Entre las funcionalidades que incluye *Coq* como asistente de programación, podemos citar:

- *Definición de tipos inductivos.* Es posible definir relaciones y tipos de datos inductivos, los cuales especifican la existencia de construcciones matemáticas concretas, como por ejemplo: desigualdades, predicados de pertenencia (relaciones) y, listas, árboles, números enteros (tipos de datos).

  Una vez definidos los tipos se pueden definir funciones, recursivas o no, y especificar propiedades sobre estos tipos, como así también probar las propiedades (eventualmente por inducción).

- *Construcción y verificación de programas funcionales*. Es posible especificar sistemas y extraer programas funcionales a partir de pruebas. Asimismo probar la corrección de programas con respecto a una especificación dada.

- *Definición y verificación de programas imperativos*. Se puede probar la corrección de programas imperativos razonando en lógica de Hoare. Sin embargo este asistente no permite, a la actualidad, derivar un programa imperativo en base a una especificación [7].

- *Definición de tipos con objetos infinitos*. *Coq* permite definir tipos de datos que pueden contener objetos no bien fundados, como por ejemplo las secuencias infinitas o *streams*, y modelar a través de estos tipos sistemas complejos, tales como: sistemas reactivos (que se comportan como una secuencia de estímulos-respuestas) y de tiempo real (sistemas reactivos cuya corrección depende de la magnitud de los retardos temporales).

En este trabajo no estamos interesados en dar una descripción completa del cálculo de construcciones inductivas y co-inductivas, ni del sistema *Coq*, en general. Por aspectos teóricos el lector puede referirse a [2, 3, 4] por el cálculo puro de construcciones, a [25] por tipos inductivos y a [5, 9, 10, 12, 17, 22] por tipos co-inductivos y aplicaciones de éstos. Acerca de *Coq*, una buena introducción son los tutoriales [11, 15] y detalles adicionales pueden encontrarse en el manual de referencia [1].

4. DESCRIPCIÓN DEL ASISTENTE COQ

A continuación desarrollamos algunas de las principales características de *Coq*, referidas en la sección previa, para ilustrar la utilidad que este asistente presenta a programadores y, en el marco de este artículo, a estudiantes de grado interesados en aprender construcción y análisis de corrección de programas funcionales e imperativos.

4.1 Definición de tipos inductivos

Coq permite definir tipos inductivos. Cada definición establece la introducción de un nuevo conjunto, junto con la manera de construir sus objetos, que además es única. Por ejemplo, podemos definir el conjunto de números naturales como sigue:

Inductive nat : Set := Co : nat | Sg : nat -> nat.

nat es el conjunto definido, en el cual sus constructores son *Co* y *Sg*. El primero representa la constante 0 del tipo *nat*, y el segundo la operación sucesor de los naturales, que dado un número n, representa el número natural n+1.

También es posible hacer definiciones inductivas paramétricas, para manejar la abstracción sobre objetos de diversos tipos. Por ejemplo, podemos definir árboles genéricos de elementos de cualquier tipo de la siguiente manera:

Inductive bintree [A:Set] : Set :=
emptyb : (bintree A) | consb : A->(bintree A)->(bintree A)->(bintree A).

bintree representa al tipo de todos los posibles árboles binarios de elementos de un tipo genérico *A*. Sus constructores son *emptyb* y *consb*, los cuales construyen el árbol vacío y un árbol binario a partir de otros dos y un elemento de tipo *A*, respectivamente.

Cuando se define un tipo inductivo, *Coq* genera tres constantes correspondientes a los principios de inducción y recursión. Las mismas implementan el principio de inducción estructural, permiten hacer definiciones recursivas y definir familias recursivas de tipos.

4.2 Definición de funciones recursivas

En *Coq* es posible definir funciones por recursión primitiva y recursión general. Un ejemplo de las primeras es la función que retorna el espejo de un árbol binario de elementos de un tipo genérico.

```
Fixpoint reverse [A:Set; b:(bintree A)] : (bintree A) :=
    Cases b of
        emptyb => (emptyb A)
        | (consb x b1 b2) => (consb A x (reverse A b2) (reverse A b1))
    end.
```

En esta definición, dado cualquier árbol binario *b* de tipo *A*, si *b* es el árbol vacío su espejo es él mismo, y si es un árbol no vacío, el espejo es el que se construye con *consb* aplicado al elemento raíz del árbol original y el espejo de sus dos subárboles permutados.

Las funciones recursivas primitivas quedan perfectamente definidas a partir de su especificación. También es posible definir en *Coq* funciones por recursión general, especificando un orden bien fundado que asegure la terminación.

4.3 Definición y prueba de propiedades

Coq permite definir propiedades sobre los tipos y los programas, y demostrarlas luego utilizando *tácticas* de prueba, en particular inducción. Una propiedad sobre los árboles binarios anteriormente definidos es: “el espejo del espejo de un árbol binario es el mismo árbol”.

Lemma rev_rev : (∀a∈Set) (∀b∈(bintree a)) (reverse a (reverse a b)) = b.

Notación. La cuantificación universal sobre un tipo S se escribe en *Coq* “(x:S) P”. Sin embargo, usaremos la notación “(∀x∈ S) P” en este trabajo para dar más claridad a las especificaciones.

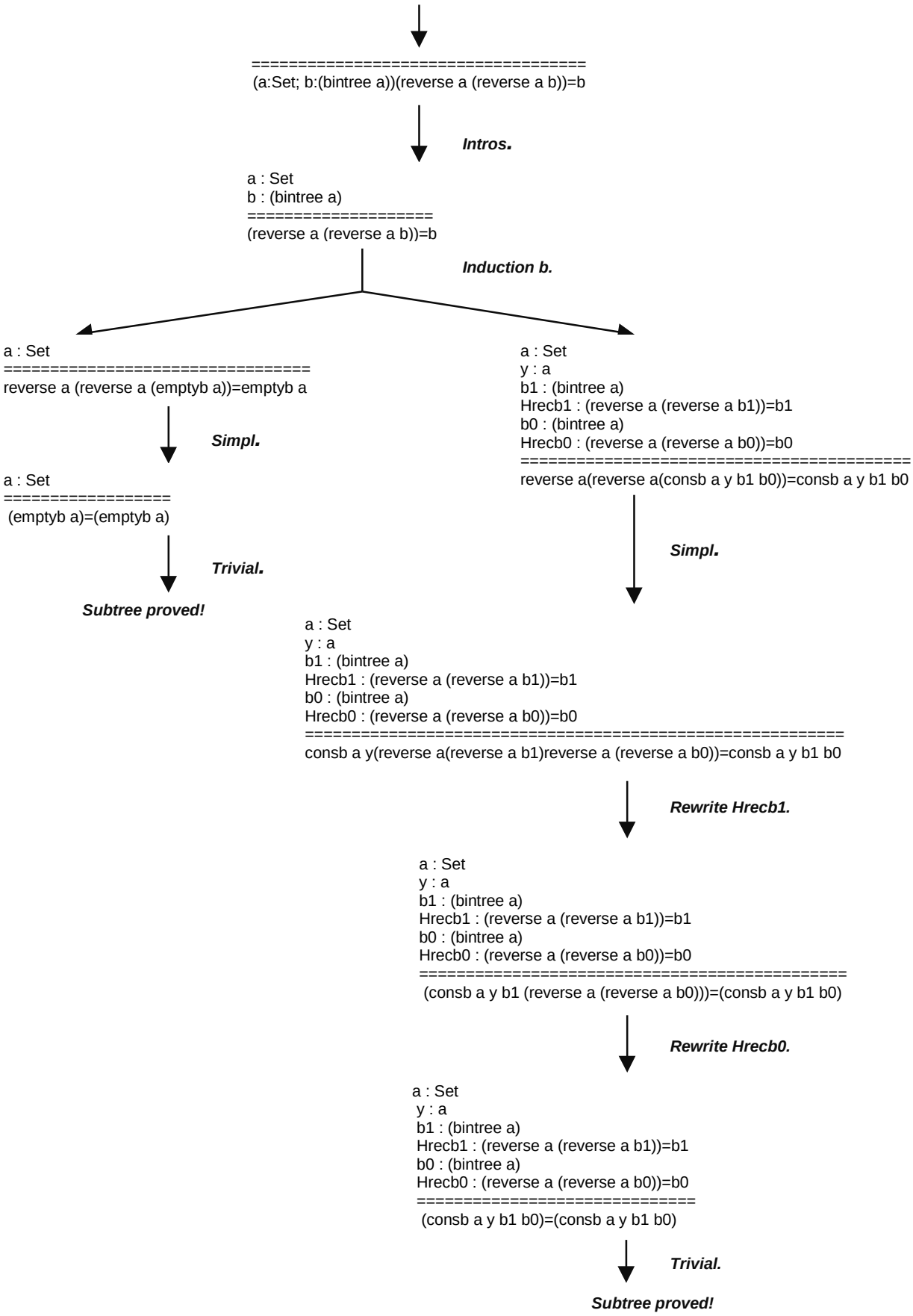
Para probar propiedades sobre un tipo inductivo *Coq* provee una táctica llamada *Induction.* Esta táctica permite usar el principio de inducción primitiva, el cual genera los casos según la definición del tipo sobre el que se establece la propiedad.

Una prueba del lema *rev_rev* en el Asistente *Coq* es la siguiente:

Proof.	Comienzo de la prueba
Intros.	Introduce las hipótesis como premisas
Induction b.	Aplica el principio de inducción estructural
Simpl.	Aplica la definición de reverse
Trivial.	Se cierra la demostración del caso base
Simpl.	Aplica la definición de reverse
Rewrite Hrecb1.	Aplica la Hipótesis Inductiva
Rewrite Hrecb0.	Aplica la Hipótesis Inductiva
Trivial.	Se cierra la demostración del caso inductivo
Save.	Culmina la demostración y salva la prueba

Gráficamente, el árbol de derivación para esta prueba –en sintaxis pura de *Coq*– es:

Lemma rev_rev : (a:Set)(b:(bintree a)) (reverse a (reverse a b))=b.



4.4 Derivación y verificación de programas funcionales

Coq permite construir funciones a partir de la demostración de un lema que establece una propiedad existencial sobre una relación. Esta relación especifica el comportamiento deseado del programa a construir. Por ejemplo, el siguiente lema especifica que para todo árbol binario existe otro que es su espejo:

```
Lemma mirror_inverse: (t:bintree) {  t':bintree | (mirror t t') }.
```

Donde el predicado inductivo *mirror* sobre árboles binarios genéricos se define como sigue:

```
Inductive mirror [A: Set]: (bintree A) -> (bintree A) -> Prop:=
    vacio: (mirror A (emptyb A) (emptyb A))
    | no_vacio: ∀x∈A , ∀t1,t2,t3,t4∈(bintree A) (mirror A t1 t2) -> (mirror A t3 t4) ->
          (mirror A (consb A x t1 t3) (consb A x t4 t2)).
```

vacio es una prueba de que *(emptyb A)* es el espejo de él mismo y *no_vacio* es una prueba de que *(consb A x t4 t2)* es el espejo de *(consb A x t1 t3)* si se tiene una prueba de que *t2* es el espejo de *t1* y de que *t4* es el espejo de *t3*.

Una vez demostrado el lema, a través de dos tácticas simplemente, podemos "derivar" (extraer) un programa en Haskell u otro lenguaje funcional, utilizando el comando *Write* disponible en la biblioteca *Extraction* de *Coq*. Para el ejemplo anterior el programa Haskell extraído es:

```
Coq > Write Haskell File "mirror_function" [ mirror_inverse].

    data Tree a = Nil | Cons a (Tree a) (Tree a)

    inverse t =  case t of
          Nil -> Nil
          Cons a0 t1 t2 -> Cons a0 (inverse t2) (inverse t1)

    mirror_inverse = inverse
```

También es posible "verificar" que un programa funcional satisface una especificación usando las tácticas *Realizer* y *Program* (o *Program_all*) en la prueba de un lema que establece una propiedad existencial sobre una relación que especifica el comportamiento deseado del programa a construir.

4.5 Especificación y verificación de programas imperativos

También es posible especificar y verificar programas imperativos en *Coq*. Por ejemplo, podemos dar el siguiente predicado inductivo binario sobre los enteros (*Z*):

```
Inductive RFact : Z -> Z -> Prop :=
    f0: (RFact '0' '1')
    | fs: ∀z,f∈Z (RFact z f) -> (RFact 'z+1'  'f*(z+1)').
```

Este predicado es válido cuando el segundo argumento es el factorial del primero. Aquí *f0* es una prueba de que *'1'* es el factorial de *'0'* y *fs* es una prueba de que *'f*(z+1)'* es el factorial de *'z+1'*, si se tiene una prueba de que *f* es el factorial de *z*. A partir del predicado *RFact* podemos especificar un programa que calcule el factorial de un número entero de la siguiente manera:

```
Global Variable f,i,n:Z ref.                          Definición de f, i y z, variables enteras
Correctness imperative_program                        Cláusula para verificar programas
{'n>=0'}                                              Precondición del programa
begin                                                 Inicio del programa
    f:=1;                                             Asignación
    i:=1;                                             Asignación
    while (!i < !n+1) do                              Comando iterativo
        {invariant  (RFact '(i-1)' f) /\ 'i<= n+1'    Invariante: f tiene el factorial de i-1
         variant 'n-i+1'}                             Cota de terminación del ciclo
        f:=!f*!i;                                     Asignación
        i:=!i+1                                       Asignación
    done                                              Fin de la Iteración
end {(RFact n@0 f)}.                                  Postcondición del programa: el factorial del
                                                      valor inicial de n (n@0) es f
```

Sintaxis: *!x* indica el valor almacenado en la variable *x*. Cabe aclarar que en especificaciones como la anterior es posible escribir funciones y relaciones utilizando notación infija (por ejemplo: `(i-1)`), al incluir una biblioteca especial de *Coq* llamada *Arith*.

Una vez definido el programa y especificado su comportamiento deseado, podemos verificar si el programa satisface la especificación, mediante la táctica *Correctness*. Esto es, si a partir de un estado de las variables que satisfacen la precondición se cumple la postcondición luego de ejecutar el programa. La demostración de fórmulas de corrección corresponde a la construcción de árboles de prueba al estilo de deducción natural. Estos árboles están construidos en términos de:

- reglas de inferencia de Floyd-Hoare para probar los objetivos (el invariante del ciclo vale al comienzo; la cota es positiva y estrictamente decreciente dentro del ciclo; el invariante se restablece luego de la ejecución un comando del ciclo; al finalizar el ciclo vale la postcondición del programa) [6, 14].
- reglas del cálculo de predicados para demostrar las fórmulas lógicas que expresan el comportamiento del programa.

4.6 Especificación y verificación de sistemas críticos: sistemas reactivos y de tiempo real

La experimentación desarrollada en el uso de asistentes de pruebas basados en teoría de tipos se ha enfocado principalmente en demostrar la corrección de programas secuenciales, pero dado su poder expresivo consideramos que pueden ser también adecuados para razonar sobre sistemas críticos, y en particular sobre sistemas reactivos y de tiempo real. Algunas experiencias llevadas a cabo en esta dirección y particularmente en *Coq* son [9, 11, 12, 17].

La formalización de estas clases de sistemas hace uso, generalmente, de tipos co-inductivos y el análisis sobre los mismos requiere por lo tanto el empleo de co-inducción como mecanismo de prueba.

Un ejemplo de tipos co-inductivos es el de las secuencias infinitas, usualmente llamadas *streams*, de elementos de un tipo dado. El tipo *streams* puede ser introducido a través de la siguiente definición:

Variable A: Set..
CoInductive Stream: Set := Cons : A -> Stream ->Stream.

La inducción estructural es la manera de expresar que los tipos inductivos sólo contienen objetos bien fundados. Aquí el principio de eliminación no es válido para tipos co-inductivos y la única regla de eliminación para streams es el análisis de casos. Este principio puede ser usado, por ejemplo, para definir los destructores head y tail.

Definition head : Stream -> A := [x:Stream] Cases x of (Cons a s) => a end.
Definition tail: Stream -> Stream := [x:Stream] Cases x of (Cons a s) => s end.

Los objetos infinitos son definidos por medio de métodos (infinitos) de construcción, semejantes a los presentes en los lenguajes de programación funcional perezosos. Estos métodos pueden ser definidos usando el comando *CoFixpoint*. Por ejemplo la siguiente definición introduce la lista infinita [a,a,a,…]:

CoFixpoint repeat: A -> Stream A := [a:A] (Cons a (repeat a)).

El comportamiento de sistemas reactivos y de tiempo real puede ser expresado en término de trazas de ejecución infinitas (*streams*) de estados, donde el paso de un estado al próximo en una traza está dado por una operación (transición) válida del sistema. Por más detalles y aplicaciones ver [5, 9, 10, 11, 12, 17, 22].

5. TALLER DE CONSTRUCCIÓN DE PROGRAMAS CERTIFICADOS USANDO COQ

En esta sección presentamos un taller para apoyar la enseñanza de métodos formales en una currícula de grado, usando el asistente de pruebas *Coq* y conceptos del área de *Teoría de Tipos*. El taller abarca fundamentalmente la especificación, derivación y verificación de sistemas en diferentes paradigmas de programación.

Nombre del taller: Construcción formal de programas en teoría de tipos.

Carácter: Grado-Postgrado.

Institutución responsable: Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay.

Objetivos

- Presentar a la Teoría de Tipos como lógica de programación y familiarizar al estudiante con ambientes de desarrollo de programas basados en este formalismo.
- Iniciar al estudiante en el uso de métodos formales para la especificación, producción, derivación y verificación de software correcto por construcción en los paradigmas de programación funcional e imperativo.
- Iniciar al alumno en el uso de métodos formales para la especificación y verificación de otras clases de sistemas. En particular, sistemas críticos. Por ejemplo sistemas reactivos y de tiempo real.
- Mostrar la utilidad de editores de pruebas basados en teoría de tipos para la especificación y verificación de aplicaciones industriales y académicas.

Temario

- Presentación formal de la lógica proposicional, de primer orden y uso de orden superior en el asistente de pruebas Coq.
- Pruebas y programas: especificaciones y tipos, su vinculación.

- Identificación de pruebas y programas. Extracción de programas a partir de pruebas. Construcción de pruebas a partir de programas.
- Recursión: definiciones inductivas, principios de inducción y esquemas de recursión.
- Construcción de programas certificados usando Coq: programas funcionales y programas imperativos.
- Extensiones: una introducción al desarrollo de programas y pruebas con tipos recursivos que pueden contener objetos infinitos. Sistemas reactivos y de tiempo real.
- Aplicaciones: análisis de desarrollos industriales y académicos realizados con Coq.

Metodología de enseñanza: Se desarrollarán clases teórico-prácticas, y se trabajará en base a tareas obligatorias que los estudiantes deberán realizar en máquina. Se utilizará como asistente de pruebas el sistema *Coq*.

Conocimientos previos recomendados: El taller presupone conocimientos previos de lógica de primer orden y de algún lenguaje de programación funcional. Asimismo, se recomienda tener conocimientos de verificación de programas imperativos y manejar nociones básicas de sistemas reactivos y de tiempo real.

Contexto en el plan de estudio de las carreras de computación del Instituto de Computación: El taller es desarrollado como asignatura electiva en los últimos semestres de la carrera de Ingeniería en Computación y como asignatura de postgrado en el marco de la maestría y el doctorado en Computación.

Modalidad de evaluación: Se seguirá un régimen de taller con tareas evaluables a lo largo del semestre. Los alumnos deberán presentar, semanal o quincenalmente, un trabajo práctico resuelto en grupos de a lo sumo dos personas. Esto le permitirá al docente evaluar gradualmente el aprendizaje de los distintos temas. Asimismo se desarrollará un proyecto final de carácter individual para integrar los temas abarcados en el taller.

Plantel docente: Un docente responsable de los contenidos teóricos, un docente responsable del taller y un ayudante por comisión en el taller.

Duración y carga horaria: El taller tendrá duración de un cuatrimestre y una carga total de 192 horas, distribuidas en horas de aula y horas de trabajo del estudiante.

Cupo: La cantidad de alumnos por comisión queda supeditada a la disponibilidad de computadoras. Estimamos conveniente un cupo máximo de 30 alumnos por comisión distribuidos en a lo sumo dos personas por computadora.

Infraestructura e insumos: Sala con 15 computadoras conectadas en red, donde se tenga instalado el software *Coq* y un PC con monocañón.

6. EXPERIENCIAS EN EL DESARROLLO DEL TALLER

Este taller viene desarrollándose regularmente en el Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República (Montevideo, Uruguay) desde el año 2000, en versiones ligeramente diferentes. Las primeras ediciones no abarcaban el estudio de sistemas críticos ni el análisis de aplicaciones industriales y académicas realizadas con *Coq*. Posteriores versiones fueron incorporando estas características y recientemente (en Febrero de 2004) anexamos al taller el módulo: análisis de aplicaciones industriales y académicas realizadas con *Coq*.

En el año 2001 desarrollamos el taller en el Departamento de Computación de la Facultad de Ingeniería de la Universidad de Río Cuarto (Argentina), y en los años 2000 y 2001 en la Universidad Nacional de Rosario (Argentina). Asimismo, versiones cortas del taller se llevaron a cabo en las siguientes escuelas de ciencias informáticas: Rio'2000, séptima escuela de verano de ciencias informáticas, desarrollada del 14 al 19 de Febrero de 2000, UNRC, Río IV, Argentina; y, ECI'2001, escuela de ciencias informáticas, desarrollada del 23 al 28 de Julio de 2001, UBA, Bs. As., Argentina. Finalmente en Febrero de 2004 presentamos una versión del taller orientada a aplicaciones computacionales –industriales– de la demostración asistida de teoremas usando Coq en el marco de la Rio'2004: décima primera escuela de verano de ciencias informáticas, UNRC, Río IV, Argentina. Aplicaciones de este tipo son por ejemplo: protocolos de comunicación, protocolos de comercio electrónico, compiladores, desarrollos de aplicaciones seguras para tarjetas inteligentes, entre otras.

En total, más de 300 estudiantes participaron de alguna edición del taller. A partir de estas experiencias surgieron cuatro tesis de maestría y varias tesinas (proyectos) de grado. Desde el año 2000 a la actualidad tres proyectos de investigación han sido llevados a cabo en el Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República (Montevideo, Uruguay) en temas relacionados y varias colaboraciones con instituciones regionales e internacionales han sido establecidas.

7. CONCLUSIONES

En este trabajo presentamos una propuesta para apoyar la enseñanza de métodos formales en una currícula de grado, usando el asistente de pruebas Coq y conceptos del área de Teoría de Tipos. El taller abarca contenidos esenciales en la formación de un profesional en Ciencias de la Computación: la especificación, construcción y verificación de sistemas en diferentes paradigmas de programación. El taller permite fortalecer la noción de que junto con la construcción de los algoritmos existe la obligación de la verificación rigurosa (formal) de su corrección y que los

programas son objetos matemáticos plausibles de ser tratados con argumentos lógico-matemáticos. Coq es una herramienta adecuada para asistir a los estudiantes en este proceso de aprendizaje.

El desarrollo de un taller de estas características permite también integrar a docentes y estudiantes interesados en trabajar en la construcción formal y la verificación de sistemas de software en diversos paradigmas de programación, desde los clásicos (imperativo y funcional) hasta otros, tales como los correspondientes a sistemas reactivos y de tiempo real. Asimismo el taller ofrece un marco adecuado para la realización de trabajos de investigación, trabajos finales en carreras de grado y trabajos de postgrados.

Actualmente se están desarrollando en el Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República (Montevideo, Uruguay) un par de trabajos –tesis de maestría– sobre el análisis de sistemas orientados a objetos usando Coq. Un trabajo futuro es extender el taller con un módulo para la especificación y la verificación de sistemas orientados a objetos en el cálculo de construcciones (co)inductivas de Coq.

Referencias

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, Ch. Murthy, C. Parent-Vigouroux, P. Loiseleur, Ch. Paulin-Mohring, A. Saïbi, and B. Werner. “The Coq Proof Assistant. Reference Manual, Versión 7.3.1”. INRIA, 2003.

[2] T. Coquand and G. Huet. “Constructions: A higher order proof system for mechanizing mathematics”. In EUROCALL´85, LNCS 203, Linz, 1985, Springer-Verlag.

[3] T. Coquand and G. Huet. “The calculus of constructions”. Information and Computation, 76(2/3), 1988.

[4] T. Coquand. “Metamathematical investigations of a calculus of constructions”. INRIA and Cambridge, University, 1986.

[5] T. Coquand. “Infinite objects in type theory”. In H. Barendregt and T. Nipkow, editors, Workshop on Types for Proofs and Programs, number 806 in LNCS, pages 62-78. Springer-Verlag, 1993.

[6] E. Dijkstra, “A Discipline of Programming”. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] J.C. Filliatre. “Proof of imperative programs”. In Chapter 18 of The Coq Proof Assistant Reference Manual, Version 7.0, 2001.

[8] A. Ferreira, C. Luna y R. Medel. “Manual-Guía de Aprendizaje de Programación Avanzada”, publicado por la Editorial de la Fundación de la Universidad Nacional de Río Cuarto –miembro de la Red de Editoriales Universitarias Argentinas– en Marzo de 1998.

[9] E. Giménez. “An application of Co-inductive Types in Coq: Verification of the Alternating Bit Protocol”. In BRA Workshop on Types for Proofs and Programs (TYPES’95), LNCS 1158, pages 135-152, Springer-Verlag, 1995.

[10] E. Giménez. A Calculus of Infinite Constructions and its application to the verification of communicating systems. PhD thesis, Ecole Normale Supérieure de Lyon, 1996, Unité de Recherche Associée au CNRS No. 1398, 1996.

[11] E. Giménez. A tutorial on recursive types in Coq, Technical Report 0221, INRIA, 1998.

[12] E. Giménez. “Two Approaches to the Verification of Concurrent Programs in Coq”, 1999.

[13] M. Gordon. Introduction to HOL: a theorem proving environment based for higher order logic. Cambridge University, Press, 1993.

[14] D. Gries. The science of programming, Springer-Verlag New York Inc., 1981.

[15] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant version 6.1, A tutorial, 1996.

[16] W. Howard. “The formulae-as-types notion of construction”. In J. Seldin and J. Hindley, editors, To H. Curry: Essays on combinatory logic, lambda calculus and formalism. Academic Press, 1980. A reprint of an unpublished manuscript from 1969.

[17] C. Luna. Especificación y análisis de sistemas de tiempo real en teoría de tipos. Caso de estudio: the railroad crossing example. Master thesis, Technical Report 00-01, InCo, PEDECIBA Informática, Fac. de Ingeniería, U. de la República, Uruguay, Febrero de 2000. Disponible también en: http://www.fing.edu.uy/~cluna.

[18] C. Luna, P. Martellotto, M. M. Novaira. “Teoría de Tipos y Coq en la Enseñanza de Programación Funcional e Imperativa. Taller de Construcción Formal de Programas”. En proceedings de la Conferencia Latinoamericana de Informática: CLEI'2002 – CIESC’2002. Montevideo, Uruguay, 2002.

[19] Z. Luo and R. Pollack. “Lego proof development system: User’s manual”. Technical Report ECS-LFCS-92-211, LFCS, 1992.

[20] L. Magnusson. The implementation of ALF – a proof editor based on Matin Löf’s Monomorphic Type Theory with Explicit Substitution. PhD thesis, Chalmers University of Göteborg, 1994.

[21] D. Mandrioli, Carlo Ghezzi, and Mehdi Jazayeri. Fundamentals of Software Engineering. Prentice Hall, 1991.

[22] L. Paulson. “Co-induction and Co-recursion in Higher-order Logic”. Technical Report 304, Computer Laboratory, University of Cambridge, 1993.

[23] C. Paulin-Mohring. “Extracting Fω’s programs from proofs in the calculus of constructions”. In Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, 1989. ACM.

[24] C. Paulin-Mohring. Extraction de programmes dans le calcul des constructions. Thèse de doctorat, Université de Paris VII, 1989.

[25] C. Paulin-Mohring. “Inductive definitions in the system Coq – rules and properties”. In M. Bezem and J. Groote, editors, Proceeedings of the conference Typed Lambda Calculi and Aplications, LNCS 664, 1993.

[26] P. Weis et X. Leroy. “Le Langage CAML”. Second edition, Dunod, Paris, 1999. First edition, InterEditions, Paris, 1993.