Teoría de Tipos y *Coq* en la Enseñanza de Programación Funcional e Imperativa. Taller de Construcción Formal de Programas[§]

Paola Martellotto, Maria Marta Novaira
Departamento de Computación, Universidad Nacional de Río Cuarto, Argentina
Ruta 8 y 36 km 601, CP:5800, Río Cuarto, Córdoba, Argentina
pmartellotto@exa.unrc.edu.ar, *mnovaira@dc.exa.unrc.edu.ar*

Carlos Luna
Departamento de Computación, Universidad de la República, Uruguay
PEDECIBA Informática, Casilla de Correo 16120, Montevideo, Uruguay
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, derivación y verificación de sistemas en los paradigmas de programación funcional e imperativo, que puede también ser adaptado a sistemas reactivos y de tiempo real. Describimos una primera experiencia en el desarrollo del taller y planteamos posibles cambios y extensiones.

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

Abstract

In this paper we introduce a proposal for teaching formal methods in an undergraduate curricula, using the Coq proof assistant and type theory concepts. We propose a course of specification, derivation and verification of systems in the functional and imperative paradigms, and an extension to reactive and real time systems. We describe too a first experiment.

Keywords: Programming Education, Type Theory, Coq, Functional Programming, Imperative Programming, Reactive and Real Time Systems.

[§] Una versión preliminar de este trabajo fue presentada en el IX Ateneo de Profesores Universitarios de Computación (IX APUC) [2].

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 definición y verificación de los 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 [9].

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, derivación y verificación de sistemas en los paradigmas de programación funcional e imperativo.

La estructura del artículo es como sigue. En la sección 2 describimos sucintamente las metodologías actualmente 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 destacar la utilidad de *Coq* como asistente para programadores. En la sección 5 presentamos nuestra propuesta, el taller de construcción de programas certificados usando *Coq*. En la sección 6 exhibimos una primera experiencia en el desarrollo del taller y finalmente, en la sección 7, incluimos las conclusiones de este trabajo.

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* [3, 3, 5], las cuales han sido formuladas como fundamento de la Matemática Constructiva. Ejemplos de estos sistemas son *ALF* [20], *Coq* [1] y *LEGO* [18]. 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 [4, 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 [17] y está basada en que las pruebas en deducción natural pueden ser representadas como términos de un cálculo lambda tipado [17], 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 [8].

- *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 [3, 4, 5] por el cálculo puro de construcciones, a [25] por tipos inductivos y a [6, 10, 11, 13, 19, 22] por tipos co-inductivos y aplicaciones de éstos. Acerca de *Coq*, una buena introducción son los tutoriales [12, 16] 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.

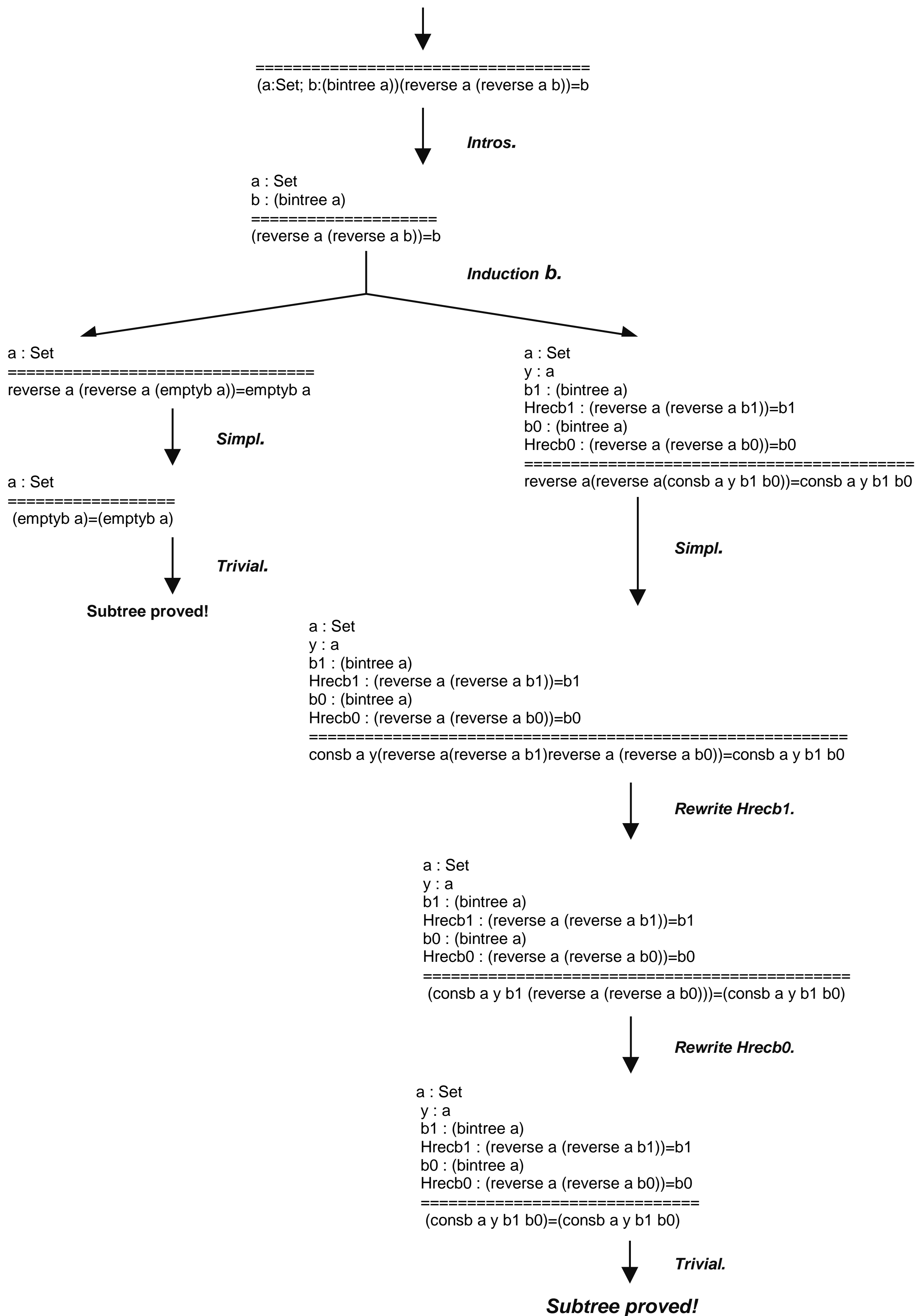
Notación. La cuantificación universal sobre un tipo S se escribe en *Coq* “(x:S) P”. Sin embargo, usaremos la notación “(∀x∈S) P” en este trabajo para dar más claridad a las especificaciones.

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:



4.4 Construcción 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 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
```

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:

Código	Comentario
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) [7, 15].
- reglas del cálculo de predicados para demostrar las fórmulas lógicas que expresan el comportamiento del programa.

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 los paradigmas de programación funcional e imperativo.

Nombre del taller

Construcción formal de programas en teoría de tipos con *Coq*.

Carácter

Grado-Postgrado.

Institutución responsable

Departamento de Computación, Fac. de Ciencias Exactas, Universidad Nacional de Río Cuarto, Argentina.
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 el paradigma de programación funcional.
- Iniciar al alumno en el uso de métodos formales para la especificación y verificación de programas en el paradigma de programación imperativo.
- Iniciar al alumno en el uso de métodos formales para la especificación y verificación de otras clases de sistemas. Por ejemplo, sistemas reactivos y de tiempo real.

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.

Metodología de enseñanza

Se desarrollarán clases teóricas y 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*. La modalidad de trabajo de los alumnos será en grupos de a lo sumo dos personas. Las clases se desarrollarán con el apoyo de dos profesores. Uno tendrá a su cargo el desarrollo de la teoría, mientras que el otro asistirá a los alumnos en la realización de los trabajos del taller. Se dispondrá de un ayudante por comisión en la parte de taller y se estipularán horarios de consulta de una hora semanal.

Conocimientos previos recomendados

El taller presupone conocimientos previos de lógica de primer orden y de algún lenguaje de programación funcional.

Contexto en el plan de estudio de las carreras de computación de la Universidad Nacional de Río Cuarto (UNRC), Argentina

El taller puede ser desarrollado en el segundo semestre del segundo año de las carreras de computación de la UNRC o eventualmente en algún semestre posterior. Los conocimientos mínimos exigidos incluyen una materia de primer año (Lógica) y una del primer semestre de segundo año (Programación Avanzada). La materia Lógica es un curso standard de Lógica de Primer Orden, mientras que en Programación Avanzada se estudia la construcción formal de programas en el paradigma imperativo y se desarrollan contenidos introductorios de Programación Funcional [9]. El taller también puede ser tomado como una asignatura optativa (electiva) de la carrera en los últimos años, o como un curso de postgrado en el área de métodos formales.

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 y/o se evaluará con un examen final.

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 una duración de un cuatrimestre y una carga de 160 horas, distribuidas en horas de aula y horas de trabajo del estudiante. La carga horaria semanal se divide en 2 horas de teórico, 3 de práctico-consulta y 5 de trabajo por parte del alumno.

Cupo (cantidad de alumnos)

La cantidad de alumnos por comisión queda supeditada a la disponibilidad de computadoras. Estimamos conveniente un cupo máximo de 30 alumnos por comisión y un mínimo de 10, distribuidos en no más de dos personas por computadora.

Infraestructura e insumos

Sala con 15 computadoras conectadas en red, donde se tenga instalado el software *Coq*; impresora, preferentemente conectada a la red; y, PC con monocañon.

6. Experiencias en el desarrollo del taller

En el segundo semestre de 2001 desarrollamos una primera edición del taller en la UNRC con el apoyo de docentes de Uruguay. En este período se realizó en paralelo la formación de tres docentes de la UNRC, que hicieron el taller y participaron como ayudantes en el mismo, brindando consultas a los demás estudiantes.

Realizamos una encuesta a los estudiantes al finalizar el curso obteniendo, entre otros, los siguientes comentarios:

- Adquirimos destrezas para especificar problemas y extraer programas correctos por construcción a partir de una especificación, así como también desarrollamos habilidades para verificar programas.
- Fortalecimos la idea de: "programar rigurosamente sobre la base de argumentos matemáticos". Esto es que, junto con la construcción de programas, garantizar la certeza de corrección es fundamental y por lo tanto es muy importante la verificación rigurosa de que un programa resuelve un problema determinado. En este punto, también fortalecimos la idea de que la inducción matemática y la deducción lógica son una buena combinación para definir, especificar y verificar programas.
- Integramos conocimientos adquiridos en las materias de la carrera, principalmente en Lógica y Programación Avanzada. En la primera, utilizamos deducción natural para demostrar teoremas. En el taller de *Coq* probamos teoremas con el mismo mecanismo que se vio en lógica pero guiados por el asistente de pruebas *Coq*, que provee tácticas (reglas de inferencias) para realizar pruebas. En la materia Programación Avanzada estudiamos como probar que un programa es correcto, tanto en el paradigma funcional como el imperativo. En el taller de *Coq* vimos como los mecanismos de prueba asociados a cada paradigma están formalizados, permitiéndonos derivar y verificar programas con el asistente *Coq*. Esto es, vimos que *Coq* es un asistente adecuado tanto para lógicos y matemáticos, como para programadores.

- Nos familiarizamos con un ambiente de desarrollo de programas que tiene a la Teoría de Tipos como lógica de programación.

En la misma encuesta los estudiantes plantearon sugerencias para mejorar el taller de *Coq*. A continuación mencionamos algunas de éstas,

- Agregaríamos como modalidad de evaluación el proyecto final de carácter individual para integrar los temas vistos en el curso, que no se llevó a la práctica por falta de tiempo.
- Seria bueno también ver una introducción al tema de sistemas reactivos y de tiempo real, dedicándoles al menos una práctica en el taller. Estos temas no fueron desarrollados por falta de tiempo.
- Los últimos prácticos resultaron muy largos y el tiempo para resolverlos fue insuficiente. Sugerimos rever el cronograma de entregas.
- Los teóricos fueron comprensibles, aunque al final no fueron suficientes para los temas que trataban.
- Se detectó la necesidad de reforzar el concepto de especificación formal en los primeros años de la carrera.

Las conclusiones de los docentes de esta experiencia serán resumidas en la próxima sección.

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, derivación y verificación de sistemas en los paradigmas de programación funcional e imperativo. 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.

Una versión similar de este taller se está desarrollando desde 1999 en el Instituto de Computación de la Facultad de Ingeniería en la Universidad de la República, Uruguay, con el apoyo de uno de los autores de este artículo. A partir del segundo semestre de 2002 desarrollaremos el taller en la UNRC, Argentina, con el apoyo de docentes de Uruguay y docentes de la UNRC. Una primer experiencia en esta institución fue realizada en 2001, donde se formaron tres docentes de la UNRC y se logró armar un equipo de gente interesada en trabajar en estos temas, tanto en Uruguay como en Argentina. En el plan de estudios de las carreras de computación de la UNRC el taller estará enmarcado como una asignatura electiva que podría ser realizada por alumnos que tengan aprobado el segundo curso de programación (*programación avanzada*, donde se estudia la construcción formal de programas en los paradigmas funcional e imperativo) y el curso de *lógica clásica*, que se desarrolla en el primer año de las carreras de computación.

El desarrollo de un taller de estas características permitirá también integrar a docentes y ayudantes 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 está analizando la posibilidad de desarrollar un sistema certificado para el frenado de vehículos en el marco de un proyecto de la UNRC.

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 7.0”. *INRIA*, 2001.

[2] M. Bongiovanni, C. Luna, P. Martellotto, M. Novaira. “Teoría de Tipos y *Coq* en la Enseñanza de Programación Funcional e Imperativa”. En IX Ateneo de Profesores Universitarios de Computación (IX APUC), El Calafate, Argentina, 2001.

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

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] E. Giménez. “Two Approaches to the Verification of Concurrent Programs in *Coq*”. To appear, 1999.

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

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

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

[17] 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.

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

[19] 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.

[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 F_ω*’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.