FUN: una herramienta didáctica para la derivación de programas funcionales

Araceli Acosta[1], Renato Cherini[1], Alejandro Gadea[1], Emmanuel Gunther[1], Leticia Losano[2], and Miguel Pagano[2]

[1] FaMAF - Univ. Nacional de Córdoba
{aacosta,cherini,gadea,gunther}@famaf.unc.edu.ar
[2] FaMAF y CONICET
{losano,pagano}@famaf.unc.edu.ar

1. Introducción

Existen diversas razones que nos llevan a reflexionar sobre las problemáticas del aprendizaje de la programación en contextos universitarios y de la formación universitaria de los profesionales de la programación. Por un lado, en el presente contexto donde el sector TIC tiene una incidencia cada vez mayor sobre el PBI nos debemos preguntar sobre las capacidades de la industria del software para brindar soluciones de calidad y proveer garantías de la corrección de los productos; en particular teniendo en cuenta la creciente importancia de métodos formales en la industria [6]. En este sentido es importante reflexionar sobre cómo la currícula de las carreras de computación ayuda a desarrollar las habilidades necesarias que permitan a sus estudiantes y egresados comprender las bases teóricas y utilizar las herramientas prácticas que permiten producir software confiable.

Por otro lado, la obligatoriedad de la educación secundaria da lugar a una población de ingresantes a las carreras de computación más heterogénea que en otras épocas, con distintos recorridos escolares, en particular en lo que respecta a las habilidades lógico-matemáticas. Esto nos presenta el desafío de lograr una mayor permanencia de los estudiantes en las carreras manteniendo el nivel de excelencia y calidad. La incorporación de tecnologías de la información a la enseñanza de los conceptos básicos de lógica y programación puede ser de gran utilidad para este propósito.

En este trabajo se describe la utilización de una herramienta desarrollada para la enseñanza de lógica y programación. En la sección 2 se presentan el contexto de utilización de la herramienta y se aborda la perspectiva didáctica; en la sección 3 se describen los conceptos básicos a enseñar con esta herramienta. En la sección 4 se describe la utilización de la herramienta con ejemplos. En la sección 5 cerramos con las conclusiones de nuestro trabajo.

2. Una perspectiva para enseñar programación

El primer curso de programación en la carrera de la Lic. en Cs. de la Computación de Fa.M.A.F es “Introducción a los algoritmos” y se dicta en el primer

semestre de primer año. El objetivo de este curso es introducir la programación en pequeña escala como la transformación de especificaciones formales en programas ejecutables; las habilidades que se espera los estudiantes adquieran son la capacidad de modelar problemas formalmente, el uso de la lógica como herramienta para razonar sobre y probar la corrección de programas. El dictado de este curso tiene una interesante experiencia acumulada durante diez años y fruto de ello es el libro [2], utilizado como material principal durante el semestre. Esta experiencia también se traduce en ciertas miradas reflexivas sobre el curso.

Desde 2007 algunos investigadores han desarrollado investigaciones focalizadas en este curso [5, 3] que permitieron comprender mejor los procesos de aprendizaje involucrados en el mismo y delinear algunas de las dificultades experimentadas por los estudiantes. A partir de entrevistas a estudiantes, en [3] se rescatan algunas percepciones de los estudiantes: (I) una importante discontinuidad entre los contenidos y las formas de estudio propias de la escuela secundaria y las de la universidad, en particular este curso; (II) dificultades, particularmente en los primeros meses, para mantener el ritmo de estudio; (III) frecuentes impedimentos para avanzar en la resolución de los ejercicios debidas a la falta de conocimiento de los detalles del lenguaje de programación.

En [5] se analizó cómo los alumnos iban construyendo demostraciones formales, qué estrategias empleaban y qué recursos utilizaban; develando que "la construcción de una prueba no sigue el carácter lineal que posee una prueba una vez finalizada". Para los estudiantes elaborar una demostración era un proceso con idas y vueltas donde se recurría a compañeros y profesores y se utilizaban artefactos, principalmente el listado de axiomas del cálculo. El proceso implicaba realizar cálculos auxiliares, probar distintos caminos –algunos infructuosos– y la discusión con los colegas del curso.

Teniendo en cuenta estas dificultades, se inició el proyecto Theona [1] con el objetivo de desarrollar material pedagógico y herramientas didácticas informáticas que faciliten los procesos de aprendizaje de los alumnos en el curso. En estos dos años se desarrolló, financiados parcialmente por Fonsoft, el material de texto y cuatro herramientas didácticas:

Equ permite realizar demostraciones, automáticamente verificadas, de fórmulas ecuacionales.

Sat ayuda a comprender el significado de fórmulas lógicas de primer orden que involucren cuantificadores a través la adecuación de mundos de objetos geométricos y propiedades de dichos modelos expresados en fórmulas lógicas.

Fun permite derivar programas funcionales a partir de especificaciones formales; y también verificar la corrección de programas.

Hal es un asistente de construcción de programas imperativos con el uso de sistemas asercionales.

2.1. Introducción a la programación funcional

La curricula a abordar con la herramienta consta de dos unidades temáticas: introducción a elementos de lógica y especificaciones y derivación de programas

funcionales. El motivo de esta elección es que el proceso de desarrollo de software en la pequeña escala puede basarse en la especificación del problema en una fórmula lógica (complicada, por cierto) y que a partir de esa fórmula se pueden realizar manipulaciones simbólicas para obtener otra expresión formal: el programa que resuelve en forma algorítmica el problema especificado en la fórmula. Otra perspectiva que admite este curso es la verificación de programas: dada una especificación y un programa (digamos programado por otro estudiante), cómo podemos convencernos que el mismo satisface su especificación. Para ello, podemos utilizar la misma maquinaria lógica para demostrar que el programa efectivamente hace lo que la especificación prescribe.

Lógica para especificaciones La primera unidad temática está dedicada a los elementos de la lógica necesarios para poder escribir especificaciones. El énfasis está puesto en los elementos que componen un lenguaje formal, primero a través de la introducción de un lenguaje de expresiones aritméticas, para pasar luego a un lenguaje de fórmulas y el sistema de pruebas de esta lógica. Notemos que en este contexto el estudiante se enfrenta por primera vez a ciertos elementos que son comunes a cualquier lenguaje de programación: constantes, operadores y su aridad, variables, un elemental sistema de tipos y una gramática que definen las frases válidas del lenguaje.

Pensamos que esta primera aproximación a los fenómenos sintácticos en una lógica proposicional tiene la ventaja de no lidiar directamente con un lenguaje de programación. La elección del lenguaje de expresiones aritméticas tiene como ventaja que uno puede explicar informalmente la semántica de las expresiones (y de las fórmulas) sin necesidad de introducir nociones de modelos; la noción de pruebas es, básicamente, una secuencia de ecuaciones justificadas por proposiciones (ya sean axiomas, teoremas o hipótesis), donde cada paso ecuacional es correcto si los términos de la ecuación coinciden sintácticamente con la justificación. También en este momento se discuten las nociones de satisfacción y validez, nuevamente pensando en la semántica natural de las expresiones aritméticas.

En una segunda etapa se extiende la lógica proposicional a lógica de predicados tipada; ésta incluye cuantificadores lógicos y aritméticos, tales como la sumatoria, la productoria y el conteo. En este pasaje aparecen nuevos conceptos sintácticos que también son comunes a lenguajes de programación, en particular la noción de ocurrencias libres y ligadas de variables, renombre de variables ligadas, sustitución.

Con las expresiones cuantificadas se introducen las especificaciones de funciones que, a través de razonamientos ecuacionales, serán transformados en expresiones del lenguaje de programación funcional.

Programación funcional El lenguaje de programación, descrito en la sec. 3.1, permite la declaración de funciones a través de constantes, operadores, pattern-matching y recursión mutua. La noción de computación en este tipo de lenguajes se basa en la reducción de expresiones hasta llegar a los valores, o formas canónicas. Este paradigma de programación tiene la ventaja de ser cercano a la práctica matemática que se tiene habitualmente, en tanto el orden de las reducciones no afecta el valor final de la computación, puesto que no hay una manipulación

de un estado. Otra buena razón, a nuestro entender, para utilizar lenguajes de programación funcionales para introducir las nociones elementales es la ausencia de variables de estado, en este sentido las variables son realmente variables matemáticas cuyo valor no varía temporalmente.

3. Conceptos básicos

En esta sección daremos un descripción general de los conceptos a abordar mediante la utilización de la herramienta. Estos conceptos se fundamentan en lo descripto en la sección anterior.

3.1. Lenguaje de programación

El lenguaje de programación que utilizaremos es un lenguaje funcional que una sintaxis similar a Haskell. Este lenguaje incluye expresiones aritméticas y expresiones booleanas proposicionales. Por ejemplo, se puede definir la función de elevar al cuadrado de la siguiente manera

$$\texttt{cuadrado} : Int \to Int$$
$$\texttt{cuadrado}.x = x * x$$

La programación funcional se funda, sobre todo, en la definición de funciones sobre tipos inductivos; para lo cual se utilizan las definiciones recursivas. Los tipos inductivos que incluye el lenguaje son los naturales y las listas. El lenguaje, al igual que Haskell, permite definiciones utilizando patrones (pattern matching). Por ejemplo para sumar todos los elementos de una lista y teniendo en cuenta que las listas son tipos inductivos, definidos a partir de un caso base y un caso inductivo, la función `sum` puede ser definida utilizando esa estructura.

$$\texttt{sum} : [Int] \to Int$$
$$\texttt{sum}.[] = 0$$
$$\texttt{sum}.x \triangleright xs = x + sumar.xs$$

Existen una serie de operadores predefinidos para las expresiones aritméticas, booleanas y para operar sobre listas. Ejemplos de operadores sobre listas son concatenar dos listas ($+\!\!+$), calcular la cantidad de elementos de una lista ($\#$), etc.

A cada definición que hemos dado, la acompañan expresiones que determinan el tipo de la función. La utilización de un sistema con chequeo estático de tipos ayuda a detectar errores, pero a la vez ayuda a la legibilidad de la función. Por otro lado, desde el punto de vista didáctico, el tipado de expresiones y funciones es una herramienta importante con la que cuentan los estudiantes para la elaboración de nuevas funciones.

3.2. Sistema formal

El objetivo de un sistema formal es explicitar un lenguaje en el cual se realizarán demostraciones y las reglas para construirlas. Esto permite tener una noción muy precisa de lo que es una demostración, así también como la posibilidad de hablar con precisión de la sintaxis y la semántica de las expresiones y fórmulas del lenguaje. En una materia introductoria a la programación estos conceptos son importantes, dado que un lenguaje de programación es un sistema formal.

El diseño de este sistema formal permite, por un lado, demostrar la corrección de programas y derivar programas a partir de su especificación, ya que el lenguaje de programación está incluido en el mismo; y por otro lado, nos permitió desarrollar una herramienta automática de verificación de demostraciones y derivaciones de los programas.

Además de las expresiones aritméticas, booleanas proposicionales y sobre listas incluidas en el lenguaje de programación, se agregan las expresiones cuantificadas; que son de mucha utilidad para la especificación de programas y propiedades. Una expresión cuantificada tiene la forma $\langle \bigoplus x : R.x : T.x \rangle$ que se entiende como $T.x_0 \oplus T.x_1 \oplus T.x_2 \oplus \ldots$ donde x_i satisface el predicado R. Los cuantificadores que utilizados son el para todo ($\forall$), el existe ($\exists$), la sumatoria ($\sum$), la productoria ($\prod$) y el contador (N).

Para especificar, por ejemplo, la función `sum` que se definió recursivamente en la sección anterior, se puede describir de la siguiente manera

$$\texttt{sum.xs} = \langle \sum i \ : \ 0 \leqslant i < \#.xs \ : \ xs.i \rangle$$

3.3. Reglas de inferencia y demostraciones

Para completar el sistema formal se definen una serie de axiomas y teoremas básicos, demostrados a partir de los axiomas, y un sistema deductivo basado en la lógica ecuacional al estilo de Dijkstra, que utiliza la transitividad y la regla de Leibniz como reglas de inferencia, y se introduce el método inductivo cuando están implicados tipos inductivos.

Los axiomas sobre los que se trabaja son un conjunto de fórmulas para la lógica proposicional y para la lógica de cuantificadores. Para la aritmética se axiomatizan algunas propiedades sobre los números naturales, necesarias a veces para demostraciones inductivas sobre naturales, y otras propiedades se dejan a criterio del estudiante, es decir, se pueden utilizar como paso en una demostración, pero no se verifican formalmente.

Una *demostración* dentro del sistema formal consiste en probar la validez de una fórmula mediante una serie de pasos justificados con axiomas y teoremas ya demostrados. A continuación se muestra un ejemplo particular de la aritmética:

$$\begin{array}{ll} & (x > 0) \lor (x \leqslant 0) \\ \equiv & \{ \text{ Aritmética } \} \\ & (x > 0) \lor \neg(x > 0) \\ \equiv & \{ \text{ Tercero excluido } (P \lor \neg P \equiv True) \ \} \\ & True \end{array}$$

3.4. El proceso de construcción de programas

En la actualidad es ampliamente aceptado que el proceso de construcción de programas debe dividirse en al menos dos etapas: la etapa de **especificación** del problema y la etapa de desarrollo del programa o de **programación**.

El resultado de la primera etapa es una *especificación formal* del problema, la cual sigue siendo abstracta (poco detallada) pero está escrita formalmente. La segunda etapa da como resultado un programa y una demostración de que el programa es *correcto respecto de la especificación dada*. A esta demostración se la llama verifiación.

Por ejemplo, a continuación se muestra la demostración de la corrección de la función `sum` cuya especificación y definición recursiva se introdujo anteriormente.

La demostración consiste en una prueba que, para cualquier lista xs, la función `sum` satisface la especificación. Esta demostración se puede hacer a partir de los axiomas y teoremas básicos del formalismo y utilizando el principio de inducción. El *caso base* supone que la lista es vacía. En esta situación la especificación se convierte en $\sum.[\,] = \langle \sum i : 0 \leqslant i < \#[\,] : [\,]!i \rangle$ que se demuestra trivialmente. Para el *caso inductivo* la especificación toma la forma:

$$sum.(x \triangleright xs) = \langle \sum i : 0 \leqslant i < \#(x \triangleright xs) : (x \triangleright xs)!i \rangle$$

que se demuestra a continuación

$$\begin{array}{ll} & \langle \sum i : 0 \leqslant i < \#(x \triangleright xs) : (x \triangleright xs)!i \rangle \\ = & \{ \text{ Definición de } \# \ \} \\ & \langle \sum i : 0 \leqslant i < 1 + \#xs) : (x \triangleright xs)!i \rangle \\ = & \{ \text{ Teorema separación del primer término } \} \\ & (x \triangleright xs)!0 + \langle \sum i : 0 \leqslant i < \#xs : (x \triangleright xs)!(i+1) \rangle \\ = & \{ \text{ Definición de } ! \ \} \\ & x + \langle \sum i : 0 \leqslant i < \#xs : xs!i \rangle \\ = & \{ \text{ Hipótesis inductiva } \} \\ & x + \texttt{sum}.xs \\ = & \{ \text{ Definición de } \texttt{sum} \ \} \\ & \texttt{sum}.(x \triangleright xs) \end{array}$$

3.5. Derivación de programas

Hasta aquí se desarrollaron tres etapas para la construcción de un programa. En primer lugar, elaborar una especificación formal para el mismo. En segundo lugar, construir el programa. En tercer lugar, dar una demostración de que dicho programa satisface la especificación.

A la construcción en simultáneo del programa y de su verificación se la llama *derivación*. En este proceso se parte de una especificación formal de una función y a través de transformaciones de la expresión se encuentra la definición de la función. Esta metodología fue desarrollada a partir de los trabajos de E. W. Dijkstra.

En la sección 4 se describe un ejemplo de derivación utilizando la herramienta.

4. FUN : una herramienta didáctica

FUN es una herramienta que consiste en un IDE (entorno de desarrollo integrado) para escribir en un lenguaje de programación y de demostraciones matemáticas. Este lenguaje permite definir progamas funcionales, especificaciones de programas, realizar pruebas matemáticas utilizando lógica proposicional y de predicados, y realizar derivaciones de programas de acuerdo con los conceptos introducidos en la sección anterior.

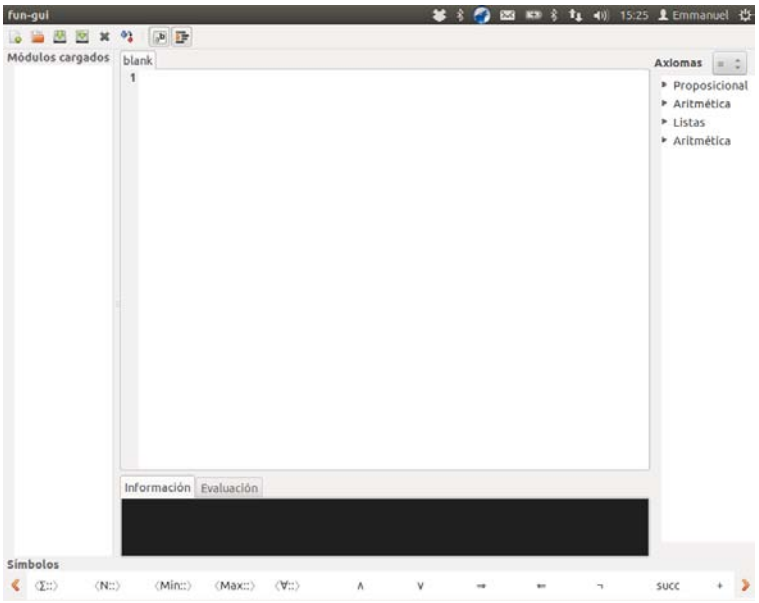

Figura 1: La pantalla principal de FUN

En la sección central de la pantalla se muestra un editor de textos, a la derecha está la lista de axiomas disponibles para utilizar como justificación en las pruebas, en la sección de abajo se ve una consola de información y otra de evaluación, y más abajo botones para poder ingresar caracteres unicode fácilmente.

El programa permite escribir y chequear pruebas del cálculo proposicional y de predicados. Las pruebas se realizan mediante la declaración de **teoremas**. Por ejemplo consideremos el teorema de doble negación $\neg\neg p \equiv p$, ver la Fig. 2b. Para realizar la prueba de $\neg\neg p \equiv p$ se utiliza un teorema auxiliar `teo1`, Fig. 2a. Para justificar los pasos de la prueba, el usuario puede utilizar una cantidad de axiomas seleccionándolos mediante la interfaz o escribiendo literalmente sus nombres, puede utilizar teoremas ya probados previamente o hipótesis, y también puede utilizar definiciones de funciones.

Una vez que se escribe un módulo (un archivo que contiene definiciones de funciones, demostraciones y/o especificaciones) el usuario puede chequear que el

```
1 module TeoremasEjemplo
2
3 let thm teo1 =
4     ¬ False ≡ True
5
6 begin proof
7   ¬ False
8   ≡ { Neutro de la equivalencia a derecha }
9   ¬ False ≡ True
10  ≡ { Negación y Equivalencia }
11  ¬ (False ≡ True)
12  ≡ { Conmutatividad de la Equivalencia }
13  ¬ (True ≡ False)
14  ≡ { Negación y Equivalencia }
15  ¬ True ≡ False
16  ≡ { Conmutatividad de la Equivalencia }
17  False ≡ ¬ True
18  ≡ { Definición de False }
19  True
20 end proof
```

(a) Teorema auxiliar

```
23 let thm dobleNeg =
24   ¬ (¬ p) ≡ p
25
26 begin proof
27   ¬ (¬ p)
28   ≡ { Neutro de la equivalencia a izquierda }
29   ¬ (¬ (True ≡ p))
30   ≡ { Negación y Equivalencia }
31   ¬ (¬ True ≡ p)
32   ≡ { Definición de False }
33   ¬ (False ≡ p)
34   ≡ { Negación y Equivalencia }
35   ¬ False ≡ p
36   ≡ {|teo1 }
37   True ≡ p
38   ≡ { Neutro de la equivalencia a izquierda }
39   p
40 end proof
```

(b) Doble negación

Figura 2: Demostraciones de teoremas

mismo sea correcto y la herramienta mostrará la lista de todas las declaraciones indicando con una tilde si la misma es correcta, Fig. 3a o con una cruz cuando no lo es, Fig. 3b. Si alguna declaración tiene un error, cuando se hace click sobre la misma en el panel izquierdo, en la consola de información se muestra un mensaje de la razón por la cual la misma no es correcta, Fig. 3c.

La derivación de funciones en el entorno FUN sigue el mismo proceso que se explicó en la sección anterior; en las Figs. 4a y 4b se muestran respectivamente los casos bases e inductivos de la derivación de la sumatoria de una lista. Como se puede ver, en ambos casos la expresión final no contiene cuantificadores y por lo tanto es un programa evaluable.

El lenguaje funcional subyacente a FUN fue definido y estudiado formalmente en [4]; utilizando la semántica operacional allí definida se implementó el evaluador integrado en el entorno. El usuario puede evaluar una expresión paso-a-paso hasta la expresión canónica.

5. Conclusiones y trabajos futuros

En el artículo hemos presentado una metodología para la enseñanza de programación enfatizando la importancia de contar con una especificación formal. El aporte más novedoso de este artículo es la herramienta FUN, que permite al estudiante realizar la derivación, programación y verificación en un único entorno. La herramienta le brinda al estudiante la posibilidad de corregir sus propias derivaciones sin necesidad de la supervisión de un docente.

En el próximo año se espera utilizar el material de estudio producido y las herramientas informáticas en el dictado del curso “Introducción a los algoritmos”.

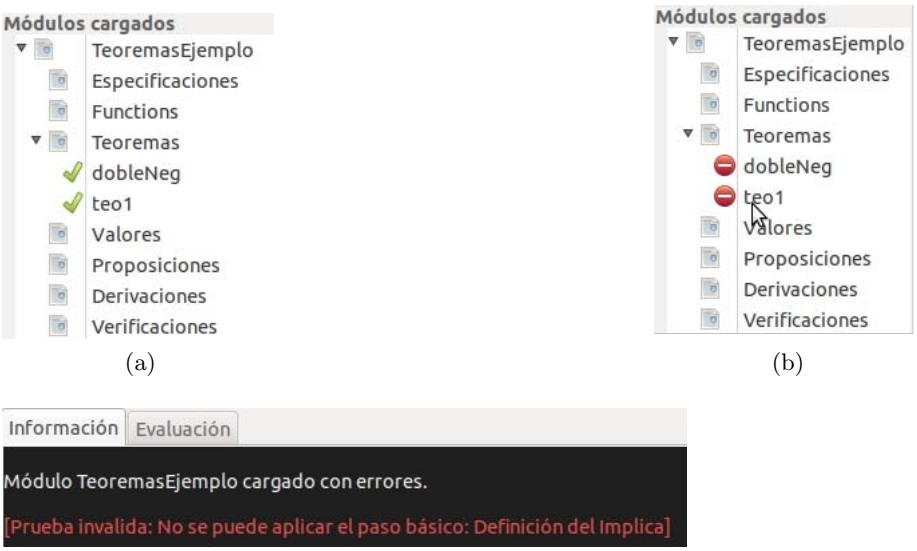

Figura 3: Módulos y sus definiciones

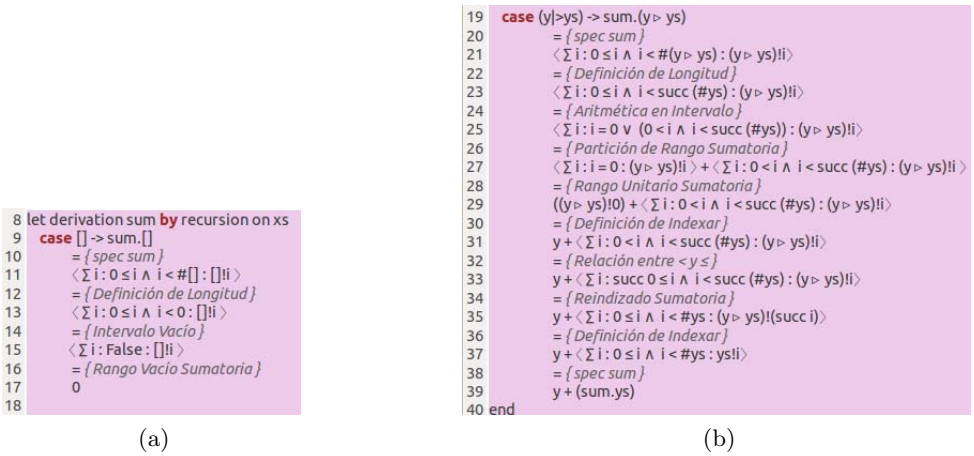

Figura 4: Derivación de `sum`

A partir del uso intensivo de las herramientas se contará con un importante feedback para mejorarlas, tanto en funcionalidades como en interfaz.

En otras líneas de trabajo se están desarrollando herramientas que complementan estos contenidos curriculares. En particular, se está desarrollando una herramienta, Hal, que persigue los mismos objetivos que FUN pero orientada a la programación imperativa; esto implica un cambio completo de paradigma. Esta herramienta permitirá completar con los contenidos conceptuales de un curso introductorio de programación desde la perspectiva formal.

Referencias

[1] Araceli Acosta y col. *Proyecto Theona*. http://www.theona.com.ar/. Mayo de 2013.

[2] Javier Blanco, Silvina Smith y Damián Barsotti. *Cálculo de Programas*. Universidad Nacional de Córdoba, 2009. ISBN: 978-950-33-0642-0.

[3] Javier Blanco y col. "An introductory course on programming based on formal specification and program calculation". En: *SIGCSE Bull.* 41.2 (jun. de 2009), págs. 31-37. ISSN: 0097-8418. DOI: 10.1145/1595453.1595459.

[4] Emmanuel Gunther. "Entorno para la Derivacion de Programas". Tesis de lic. Universidad Nacional de Córdoba - Facultad de Matemática, Astronomía y Física, jul. de 2013.

[5] Leticia Losano. "Procesos situados de aprendizaje en cursos básicos deprogramación: volverse miembro de una comunidad". Tesis doct. Universidad Nacional de Córdoba - Facultad de Filosofía y Humanidades, abr. de 2012.

[6] Mariëlle Stoelinga y Ralf Pinger, eds. *Formal Methods for Industrial Critical Systems*. Vol. 7437. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-32468-0. DOI: 10.1007/978-3-642-32469-7.