Diseño de Soluciones Abstractas mediante el Refinamiento de Especificaciones

J. M. Burgos, J. Galve, J. García, M. Sutil
Dept. Lenguajes y Sistemas Informáticos (LSIIS)
Universidad Politécnica de Madrid
28660 Boadilla del Monte - Madrid
e-mail: {jmburgos, jgalve, juliog, msutil}@fi.upm.es

Resumen

El contexto de la presente propuesta es la integración de metodos formales en la enseñanza de la programación. Aun cuando existe un amplio consenso sobre la importancia de las herramientas de descripción formal en el desarrollo de aplicaciones industriales, dicha necesidad no ha sido asumida con éxito en el curriculum de informática.

En este trabajo presentamos una metodología para la construcción de programas a partir de la especificación de los problemas. De esta manera, es la etapa de análisis donde se vertebra y organiza el conocimiento de programación que se quiere ofrecer al estudiante. El concepto de descripción precisa se incorpora gradualmente al proceso de aprendizaje: primero informalmente, a modo de documentación, y posteriormente como formalismo matemático basado en la lógica de predicados. La propuesta está particularmente enfocada a ofrecer un marco general de diseño abstracto de soluciones para problemas que manejan colecciones de datos; para ello, hemos hecho una adaptación comprensible de la técnica de diseño *"decrece y vencerás"*[2].

1. Introducción

El beneficio de aplicar metodos formales al desarrollo de sistemas informáticos desde la especificación, a través del diseño, hasta alcanzar los programas son significativos, y han sido descritos en innumerables trabajos. Sin embargo, y a pesar de las numerosas propuestas de modelos de enseñanza de la programación, por un lado, y la gran cantidad de teorías, modelos y lenguajes de descripción formal de componentes del software, hay una escasa literatura sobre la aplicación de técnicas formales para organizar el conocimiento de programación de una manera razonable y comprensible.

En nuestra propuesta, pretendemos que los estudiantes de programación sean capaces de utilizar las técnicas de especificación de manera realista, como lo haría un desarrollador experto. Para ello planteamos la utilización de esquemas de especificación que sirvan como bloques de construcción de programas, para adquirir primero y describir después, las piezas de conocimiento que utilizan cuando escriben programas. Para alcanzar este objetivo, se propone lo siguiente:

1. Una taxonomía o conjunto organizado de esquemas de especificación [1]. Esto define el primer nivel de bloques de construcción para los programas.
2. Una teoría de diseño abstracto de soluciones y algoritmos basada en la evolución del diseño de una solución mediante el refinamiento progresivo de especificaciones formales, que sirva de "plataforma de trabajo".

Todo lo anterior, debe servir a los estudiantes para alcanzar una experiencia en diseño de programas realista y correcta.

El resto del artículo se organiza como sigue. Primeramente, se presenta el modelo formal de funciones PRE/POST, que nos sirve de base para la especificación formal de problemas. En la sección 3 se describe una taxonomía de esquemas de especificación, establecida sobre el corpus de problemas estudiados en un primer curso de programación. La taxonomía utiliza como criterio

de clasificación las similitudes que puedan encontrase entre especificaciones de problemas diferentes. Posteriormente, en la sección 4, se presentan los conceptos de cuantificación y cuantificación operacional asociados con la especificación de problemas que involucran el manejo de colecciones de datos. Finalmente, se concluye presentando unos ejemplos de uso de la metodología en el desarrollo de algoritmos recursivos e iterativos para problemas de la taxonomía.

2. Especificación Formal PRE/POST

Como es bien conocido, un problema puede escribirse como una función matemática, cuyo significado viene descrito según el modelo de especificación basado en precondiciónes y postcondiciones (PRE/POST), como sigue:

```
Problema (x : D) : (resultado : R)
pre: pre (x)
post: post (x, resultado)
```

donde,

- `x` es el nombre (o nombres) de los argumentos del problema.
- `D` es el dominio de datos de entrada del problema. Puede ser un dominio simple o un producto cartesiano de dominios.
- `resultado` es el nombre (o nombres) del resultado devuelto por el problema.
- `R` es el dominio de datos de salida del problema. Puede ser un dominio simple o un producto cartesiano de dominios.
- *pre (x)* (o pre-condición) es el predicado que describe las condiciones que los argumentos de entrada deben cumplir.
- *post (x, resultado)* (o post-condición) es el predicado que describe el significado abstracto del problema, estableciendo las relaciones entre los argumentos de entrada y el resultado.

Sin querer ser muy precisos, lo que la especificación establece es aquellos requisitos concretos determinados por el problema a resolver. Una especificación se describe como un *contrato* que explica cuál es el comportamiento esperado de las soluciones. Las especificaciones pueden ser formales o informales. Por ejemplo, en el siguiente problema:

> *"Dados 2 números enteros positivos a y b (con a ≤ b), escribir un programa que devuelva la suma de los cuadrados de los números incluidos entre a y b".*

se puede plantear la siguiente especificación informal:

```
SumaCuadrados (a, b :Int):(resultado: Int)
pre:   "a es menor o igual que b, a y b
       positivos"
post:  resultado = "suma los cuadrados de
       los números en [a..b]"
```

3. Una Taxonomía de Esquemas de Especificación de Problemas

Clasificamos los problemas tomando como principio la similitud de sus especificaciones. Como principal criterio, aunque no el único, relacionamos dos problemas comparando el "aspecto" de sus post-condiciones. De esta forma, dado una especificación que responda al esquema:

```
Problema (x : D) : (resultado : R)
pre: pre (x)
post: post (x, resultado)
```

podemos encasillar una especificación dentro de alguna de las siguientes categorías [1]:

- *Solución Directa*: problemas descritos como una expresión simple, somo sigue:

  ```
  resultado = <simple expression>
  ```

- *Análisis de Casos*: problemas descritos como varios subproblemas alternativos con guardas (como, una expresión condicional):

  $$\begin{aligned} &cond_1\ (x) \rightarrow resultado = Exp_1\ (x)\ \ /\backslash \\ &\qquad\qquad ... \\ &cond_n\ (x) \rightarrow resultado = Exp_n\ (x) \end{aligned}$$

- *Cuantificaciones*: problemas descritos como una acción repetida sobre un dominio:

$$resultado = Q\ i \in \langle dom \rangle | Filtro(i).Accion(\alpha)$$

donde, la *accion* a computar depende del tipo de cuantificador. Estos esquemas de problemas pueden ser, a su vez, subclasificados en diferentes variedades dependiento de la naturaleza de la acción:

a) *Acumulación*: La acción que se computa acumula valores (p.e., especificaciones que utilizan los cuantificadores matemáticos Σ y Π).
b) *Búsqueda*: La acción consiste en evaluar una propiedad sobre uno o más elementos (p.e., especificaciones que utilizan alguno de los cuantificadores lógicos $\exists$ y $\forall$).
c) *Maximización:* La acción maximiza una expresión (p.e., el mayor número primo en una colección).
d) *Construcción*: La acción construye una colección de elementos (p.e., construir la secuencia de números primos a partir de una secuencia de números enteros positivos).

La construcción de buenos programas que resuelvan este tipo de problemas constituye una parte importante de los contenidos de un primer curso de programación. A este objetivo está dirigida gran parte de la metodología de diseño de programas que aquí se propone.

La siguiente sección está dedicada a la aplicación del refinamiento de especificaciones formales sobre la familia de esquemas de cuantificación.

4. Cuantificaciones

La resolución de problemas que involucran el manejo o manipulación de colecciones de elementos no sólo son propios de los cursos iniciales de programación, sino que constituyen la parte más importante de los mismos.

Las cuantificaciones son fórmulas matemáticas que ofrecen un soporte formal para la descripción de problemas que involucran la computación sobre una colección de elementos. En este punto, la interpretación que podemos darle a los esquemas de cuantificación es doble :

- *Declarativa*: en tanto que nos provee con una descripción abstracta del problema.
- *Algorítmica* o *computacional*: en tanto que ofrece pautas de diseño para la obtención de esquemas abstractos de solución.

Los pasos que nos permiten transformar la primera interpretación en la segunda constituyen, sin duda, una parte importante de las decisiones de diseño que debemos tomar para alcanzar una solución. Así, como resultado de varias transformaciones, acabamos alcanzando una descripción operacional o algoritmo que nos permite, casi automáticamente, obtener la implementación de un programa de forma satisfactoria.

Siguiendo con la propuesta general del artículo, nuestra intención es definir una metodología general de diseño de soluciones para el tipo de problemas que operan con colecciones. Para ello, y como táctica de descripción de problemas, proponemos la especificación formal mediante cuantificaciones, y como estrategia de diseño, el refinamiento sucesivo de las especificaciones en otras equivalentes "más operacionales". Como se mostrará en las siguientes subsecciones, un punto crucial de la estrategia se centra en la transformación de los dominios de cuantificación en otros "dominios recorribles". Esto último asegurará la operacionalidad del cuantificador.

4.1. Cuantificaciones en la Especificación de Problemas

Como se había comentado, una cuantificación puede describirse como la fórmula:

$$Q\ i \in C \mid Filtro(i)\ .\ Expresion(i)$$

cuyo significado intuitivo es:

> "*Computar repetidamente* Expresion sobre aquellos elementos del dominio C *que satisfagan el* Filtro y combinarlas con mediante el operador $\oplus$."

donde

- *Q* es el cuantificador u operador de cuantificación que representa al tipo de cuantificación. En Matemáticas se tiene el sumatorio (Σ) y el productorio (Π); en

Lógica contamos con los cuantificadores existential (∃) y universal (∀).

- *C* es el dominio cuantificado, definido a partir de los datos del problema (D).
- i es la variable de cuantificación, definida sobre el dominio cuantificado.
- *Filtro* es una expresión booleana que filtra los elementos del conjunto C.
- *Expresion* describe la acción que se cuantifica.

Otras dos componentes se hayan implicitas en una cuantificación:

- ⊕ , la operación asociada al cuantificador (+ en ∑, * en ∏, ∨ en ∃, ∧ en ∀, etc.).
- *Minimal,* el valor de la cuantificación si se aplica al caso básico del dominio cuantificado[1].

Filtro y *Expresion* son las componentes descriptivas propias de cada problema concreto, mientras que el resto lo son de la categoría de cuantificación a la que pertenecen. Como se verá posteriormente, Filtro y Expresion aparecerán en el esquema de diseño de la solución obtenida.

Tomemos como ejemplo el problema de "*sumar los cuadrados pares de una colección*"". Si suponemos que la función `EsPar` está definida, se tiene que:

```
SumarCuadradosPares (a,b : Int):(r : Int)
Pre:  0 < a ≤ b
Post: r = Σ x ∈ [a..b] | EsPar (x) . x²
```

4.2. Refinamiento de Especificaciones

En la sección anterior se comentaba que con una cuantificación estamos expresando formalmente la ejecución repetida de una acción sobre los elementos de una colección. Sin embargo, y a pesar de la idea intuitiva ofrecida, las cuantificaciones no siempre ofrecen información útil sobre cómo recorrer los dominios que se hayan cuantificados.

[1] Habitualmente, este valor será el elemento neutro para el operador ⊕ .

Por este motivo, y para añadir cierta información operacional en las cuantificaciones, necesitamos replantear la especificación de las colecciones y describirlas en función de *dominios de recorrido (DR).* Vamos a seguir un método inductivo de construcción de los dominios de recorrido. Adelantándonos al conjunto de definiciones que nos permitirá describir los de dominio de recorrido, podemos adelantar cuál será el aspecto de las nuevas cuantificaciones, como sigue:

```
Q i ∈ DR | Filtro (i). Exp (i)
```

donde:

```
DR = {Inicial(x)..Siguiente(i)..Dentro(i)}
```

y cuyo significado es:

"Computar repetidamente Exp sobre los elementos del dominio DR (modelado como el rango de valores que, comenzando en un valor Inicial avanza con Siguiente, repetidamente mientras la condición Dentro se mantiene) que satisface Filtro y los combina con el operador ⊕."

Ahora, proponemos una definición formal de los nuevos elementos de las especificaciones:

- Inicial: primer valor del dominio de recorrido *(Inicial : D → DR)*
- Siguiente: determina el siguiente valor para un valor de DR: (*Siguiente : DR → DR)*
- Dentro*:* determina si un valor pertenece a DR o no: (*Dentro : DR → Bool*)

4.2.1 Dominios de Recorrido

Las definiciones anteriores nos ofrecen una fundamentación matemática suficiente para proponer una interpretación basada en cuantificaciones para la técnica de resolución de problemas denominada *"decrementa y venceras"*[2]

[2] El principio de la teoría de problemas *"decrease-and-conquer"* consiste resolver un problema de tamaño grande reduciéndolo a un problema más pequeño y resolver este último, para después, se combina el valor resultante con alguna información del problema original. La existencia de un pre-orden bien fundado sobre el dominio del problema (i.e., un operador ≤) asegura la

[2], a la que pertenecen el tipo de soluciones habituales en un primer curso de programación.

Así, dado un problema descrito como:

Problema (x : D) **:** (resultado : R)

y especificado como una cuantificación, podemos construir inductivamente un dominio de recorrido DR a partir de D como sigue:

DR_1 = Inicial (x), si Dentro (Inicial (x))
DR_{i+1}= Siguiente (DR_i), si Dentro (Siguiente (DR_i))
DR = $\cup$ DR_i, i $\in$ {1. . ω^3 }

Es fácil comprobar que la anterior definición induce un pre-orden bien fundado, como sigue:

$\leq^{DR}$: DR $\times$ DR $\rightarrow$ Bool
$Siguiente^i$ (x) $\leq^{DR}$ $Siguiente^j$ (x) $\Leftrightarrow$ i $\leq$ j

Si ahora podemos definir una función biyectiva Inversa : DR $\rightarrow$ D se tendrá que las dos especificaciones son equivalentes, y que por tanto están describiendo el mismo problema. Por ejemplo, el problema SumarCuadradosPares, admite la siguiente interpretación:

D = Int x Int DR = Intervalos (Int)
Inicial (a , b) = [a..b] Siguiente ([x..y]) = [x+1..y]
Dentro ([x..y]) = x $\leq$ y Inversa ([x..y]) = (x, y)
Pr ([x..y]) = x

Esto nos lleva a una especificación equivalente pero con la información operacional suficiente como para recorrer el dominio cuantificado, como sigue:

```
SumarCuadradosPares (a,b : Int):(r : Int)
pre:  0 < a ≤ b
post: r = Σ x ∈ [a..b] | EsPar (x) . x²
sol:  r = Σ I ∈ DR | EsPar(Pr(I)) . Pr(I) ²
```

5. De Soluciones Abstractas a Algoritmos

Cada solución abstracta en una especificación operacional denota un conjunto de posibles implementaciones en algún modelo de cómputo.

terminación.

[3] Podría ser $\omega \neq$ Card (D)

En la actualidad nosotros nos referimos a dos posibles modelos de computación: recursividad lineal y programas iterativos. Además, debemos elegir la implementación adecuada para los tipos abstractos que aparecen en las soluciones abstractas.

Normalmente, los lenguajes de programación ofrecen una implementación estandar para los tipos básicos. Sin embargo, cuando aumenta el grado de abstracción del lenguaje, y se incluyen tipos de datos de más alto nivel (como conjuntos, secuencias y tablas) o los usuarios especifican sus propios tipos de datos, las implementaciones que denominaríamos *estandar* comienzan a ser menos satisfactorias. Dependiendo de la mezcla de operaciones, la frecuencia en que las invocamos, información sobre el tamaño utilizados y otras consideraciónes, se podría dar la circunstancia de que una implementanción pudiera ser mucho mejor que otras. Por tanto, no hay una implementación única que nos ofrezca un buen rendimiento para todas las circunstancias del tipo abstracto.

Una vez que la solución ha sido diseñada abstractamente, tenemos que unir las piezas en una estructura de algoritmo. Como una solución abstracta admite diferentes interpretaciones computacionales, debemos elegir el modelo de computo sobre el que vamos a operar. En nuestro caso, hemos elegido dos modelos de computación: el recursivo lineal y el iterativo.

5.1. Interpretación Recursiva

La interpretación recursiva de una solución abstracta para un problema del tipo:

```
Problema (x: D) : (resultado : R)
```

expresado como una cuantificación sobre un dominio de recorrido, es como sigue:

```
resultado =
  if ¬ Dentro (x)  then
       Minimal (x)
   else-if ¬ Filtro (Inverso(x))  then
       Problema (Siguiente ((x))
   else
       Expresion (Inverso (x)) ⊕
       Problema (Siguiente (x))
   fi
```

Siguiendo este esquema, podemos implementar recursivamente la función SumarCuadradosPares. Utilizando el lenguaje Haskell, sería como sigue:

```
sumaCuadradosPares (a,b) =
 | a > b         = 0
 | not esPar(a) = sumaCuadradosPares(a+1,b)
 | otherwise    = cuadrado (a) +
                  sumaCuadradosPares(a+1,b)
```

5.2. Interpretación Iterativa

La interpretación iterativa de una solución general abstracta es:

```
Problema (x: D) : (resultado : R)
(resultado, i)  := (Minimal(x), Inicial(x))
while Dentro ( i ) do
-- Invariante:
-- {resultado =
--  Q j ∈ [Inicial(x)..i)|Filtro(j).Exp(j)}
   if Filtro (i) ) then
     resultado := Expresion (i) ⊕ resultado
   fi
   i := Next ( i )
od
-- {resultado = Q i ∈ C|Filtro(i).Exp(i) }
```

Nótese que la cuantificación con dominios de recorrido permite deducir el invariante de los bucles en la implementación. Una cuestión importante que considerar es el significado de la operación abstracta ':=', que no siempre denota una asignación simple de un valor a una variable.

Para el problema SumarCuadradosPares, en el lenguaje Ada se tiene:

```
function Suma_Cuadrados_Pares
       (a, b:Integer) return Integer is
i, resultado : Integer;
begin
   (resultado, i)  := ( 0, a )
   while not (i > b) loop
     if EsPar (i) then
       resultado:= cuadrado(i) + resultado;
     else
       i := i +  1;
     end if;
   end loop;
   return resultado;
end Suma_Cuadrados_Pares;
```

6. Conclusiones y Trabajos Futuros

En este trabajo hemos presentado una propuesta inicial de enseñanza de la programación mediante diseños abstractos de soluciones. Los dos puntos clave de la propuesta son: a) la existencia de una clasificación de problema, y b) la posibilidad de construir programas mediante el refinamiento progresivo de las especificaciones. El modelo de diseño abstracto puede ser situado en una jerarquía de refinamientos con especificaciones, ofreciéndose las pistas de refinamiento que permiten obtener programas correctos, bien estructurados y basados en ideas abstractas.

En el futuro, planeamos mejorar la propuesta con más componentes de inferencia deductiva (como simplificación, evaluación parcial, etc..). Nuestra intención es ofrecer un marco de desarrollo para la transformación de especificaciones formales que puedan componerse, via información táctica de diseño o meta-programación, y que permitan definir transformaciones abstractas para dominios de problemas específicos. Más aún, una propuesta más ambiciosa debería permitir ciertas soluciones basadas en otras técnicas generales de diseño, como son el uso de parámetros de acumulación o la generalización de problemas, así como una interpretación abstracta de la técnica de refinamiento progresivo y las estrategias de composición.

Finalmente, nos atrevemos a conjeturar que la actual taxonomía de esquemas de especificación se verá ampliada en el futuro.

Referencias

[1] Burgos, J.M., Galve, J., García, J. and Sutil M.: Una Taxonomía de Problemas para la Enseñanza de la Programación, *Actas del VII INFOREDU'2000*, Mayo 2000, La Habana (Cuba).

[2] Schmidt, D.R. Towards a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, AMAST'96 (1996), vol. LNCS 1101, Springer-Verlag, pp. 62-84.