Hacia la Prueba de Corrección de Clases

Valentín Cassano
Departamento de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, Córdoba, Argentina
valentin@dc.exa.unrc.edu.ar

Damián E. Nadales
Departamento de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, Córdoba, Argentina
dnadales@dc.exa.unrc.edu.ar

Resumen

La actividad de desarrollar programas orientados a objetos, los cuales involucran referencias a memoria, pueden introducir errores difíciles de identificar con el uso de un razonamiento operacional. Esto da lugar a la necesidad de contar con un marco teórico para la prueba de corrección de clases.

A partir de esta motivación y basandonos en la idea propuesta por B. Meyer en [Meyer 03a], trabajamos en el desarrollo de una semántica formal para probar, matemáticamente, que clases equipadas con contratos satisfacen los mismos.

Keywords: Métodos Formales, Orientación a Objetos, Prueba de Corrección, Contratos, Funciones de Abstracción.

Abstract

The activity of object oriented program development, which may involve references variables, is prone to errors which are hard to find by operational reasoning. Hence the need for a theoretical framework for proving class correctness.

Given this motivation and based on ideas proposed by B. Meyer in [Meyer 03a] we develop a formal semantics for mathematically proving that contract equipped classes satisfy their specifications.

Keywords: Formal Methods, Object Orientation, Correctness Proofs, Contracts, Abstraction Functions.

1. Introducción

Tal como se lo muestra en varios trabajos ([EWD 1036], [EWD 76]), el razonamiento operacional sobre programas es una actividad no sólo mentalmente desgastante, sino también proclive a la introducción de errores. Si además se consideran aquellos programas que involucran variables conteniendo referencias a memoria, la situación es aún peor ya que aparecen casos de aliasing. Para ilustrar sobre

los problemas que acaecen cuando se trabaja con referencias a memoria, damos a continuación un ejemplo.

Supongamos que se define una clase A de la siguiente manera:

```
class A
  feature{NONE}
    f:A
    g:A
  feature{A}
    get_f is
    do
      Result := f
    end
```

Supongamos además que tenemos dentro de A un método cuya implementación se compone de las siguientes sentencias:

```
S; f := g; a := f.get_f
```

donde f y g son los atributos que arriba declaramos.

Ahora, dada esta otra secuencia de instrucciones (siendo S, f y g los mismos que en la implementación anterior):

```
S; a := g.get_f
```

Se plantea el interrogante si en ambas implementaciones el valor de a al finalizar la ejecución de las sentencias respectivas será el mismo.

En este punto quizá el lector este receloso de responder afirmativamente. Tal vez resulte más interesante preguntar por que la proposición anterior no es verdadera para el caso general. Con las herramientas desarrolladas en este trabajo se puede dar una respuesta (formal) a este interrogante.

La necesidad de una semántica para la prueba de corrección de programas se pone de manifiesto además cuando queremos probar que clases equipadas con contratos satisfacen los mismos. Pero esto no es posible si no se cuenta con un marco teórico adecuado.

Consideremos por ejemplo el método make de la clase LIST_SET la cual implementa el álgebra de los conjuntos usando listas, y make crea un objeto representando el conjunto vacío. Ahora preguntemonos cual es la postcondición de este método. Con los constructores sintácticos de Eiffel (o de cualquier otro lenguaje de programación imperativo) no es posible formalizarla, es necesario contar con un mayor poder expresivo en los contratos. Tal requerimiento es satisfecho en este trabajo, pero todavía esto no es suficiente. Suponiendo que en las aserciones se permite la utilización de funciones matemáticas abstractas y que tenemos un atributo Ds: LIST[G] el cual esta destinado a contener los elementos del conjunto, es probable que se quiera expresar la postcondición de make del siguiente modo: $Ds = \emptyset$.

Sin embargo la expresión anterior sufre dos inconvenientes. El mas leve es que es dependiente

de la implementación, un conjunto podría quizá estar representado mediante un atributo con diferente nombre, o peor aún podrá representarse simultáneamente usando un árbol y una pila, por ejemplo, por lo que mas allá del nombre la postcondición anterior no nos sirve. Entonces resulta un tanto contradictorio pensar en dos post condiciones diferentes para dos métodos cuyo comportamiento esperado es exactamente el mismo. Pero mas grave aún, la expresión anterior se encuentra incorrectamente tipada, estamos igualando un objeto del tipo `LIST` con un conjunto.

Es importante tener clases probadas, en especial aquellas que implementan álgebras como enteros, árboles, grafos, etc, porque son utilizadas exhaustivamentes por el programador de aplicación, quien construye su software basándose en estas.

Uno puede optar por omitir estas pruebas de corrección, pero trayendo consigo el riesgo de errores graves para la consistencia de los programas. El lector puede intentar analizar el siguiente método (implementado en el lenguaje `Eiffel`) que invierte el orden de los elementos de una lista e intentar determinar si el mismo cumple su propósito[1]:

```
  reverse is
 -- Change the list to have the same elements in reverse order
local previous, next: NODE[G]
do
   from
      next := first
   until
      next = Void
   loop
      previous := first
      first := next
      next := next.right
      first.put_right(previous)
end
```

El lector perceptivo, habrá notado que el error es difícil de encontrar. Pero lo que es más grave aún, es que el cliente del método asume (como es de esperar) que el mismo está correctamente implementado, mientras que para el caso considerado el simple hecho de invocar a `reverse` y luego recorrer la lista dejará a su programa en un loop infinito. Es importante notar que errores como estos pueden encontrarse en varios programas en uso.

El esfuerzo que requiere la correccion matemática de componentes de software es justificado por el amplio reuso que los mismos poseen. Y si tales piezas de software han de ser usadas en sistemas críticos, esta justificación se hace necesidad.

En el presente trabajo, se introducen las nociones básicas del formalismo con el objetivo de exponer luego la semántica definida. Además, se presenta la noción de modelo asociado a una clase con el propósito de servir como función de abstracción de forma tal de especificar los contratos de una clase, resolviendo los inconvenientes que presentamos anteriormente. El trabajo concluye con un ejemplo

[1] la clase `NODE` contiene un campo `right` que mantiene una referencia al próximo elemento de la lista. El método `put_right(x)` actualiza dicho campo con el valor referenciado por `x`

práctico, donde se hace uso del formalismo desarrollado para establecer la corrección de un método.

2. El formalismo básico

En esta sección se definen algunos de los conceptos utilizados en el presente artículo. Los mismos se introducen en [Meyer 03a] y se encuentran definidos con mas detalles en [Blanco 05], en donde además se corrigen algunos errores del trabajo original.

Habiendo introducido estos conceptos, se describirá la semántica formal asociada a un subconjunto del lenguaje de programación `Eiffel`.

2.1. Nociones preliminares

Las abstracciones aquí presentadas sirven de base para la definición de la semántica sobre la cual se realizan las pruebas matemáticas de corrección de clases equipadas con contratos al estilo `Eiffel`.

A continuación se brinda una definición de cada uno, y cual es su significado en este contexto de prueba.

- *Address*: Denota el conjunto de todas las posibles direcciones abstractas de memoria.
- *Objects*: Denota el conjunto de todas las direcciones que pueden albergar objetos. Cabe destacar que *Objects* $\subseteq$ *Address*.
- *Expanded*: Denota el conjunto de todos los valores que no son referencias; por ejemplo los enteros, los booleanos, etc.
- *Values*: Denota el conjunto de todos los posibles valores de un sistema; es decir *Values* $\doteq$ *Objects* $\cup$ *Expanded*.
- *States*: Denota el conjunto de todos los posibles estados de un sistema. Un elemento $s \in States$ se define como una tupla cuyo primer componente es un conjunto de funciones ($Objects \nrightarrow Objects$) que modela las variables involucradas en el sistema; y la segunda componente es un conjunto que contiene los objetos presentes en ese estado.

Así en este sistema formal una variable representara una función con alguno de los siguientes perfiles:

- $Objects \nrightarrow States \nrightarrow Objects$, para aquellas que constituyan una referencia a otros objetos.
- $Objects \nrightarrow States \nrightarrow Expanded$, para aquellas las cuales contengan un valor subclase de `EXPANDED`, como por ejemplo las del tipo `INTEGER`. Donde $Expanded$ es el conjunto formado por la unión disjunta de los conjuntos matemáticos $\mathbb{B}$, $\mathbb{R}$, etc.

Éste enfoque expresa el hecho de que el objeto el cuál una variable `f` referencia o el valor que ésta contiene (en el caso que `f` sea de tipo `EXPANDED`) en una instancia de la ejecución de un programa Eiffel, varia según el estado y el objeto en donde el identificador `f` se halle.

2.1.1. Interpretando cambios de estados

Un programa orientado a objetos consiste de una secuencia de cambios de estados que reflejan la ejecución de determinados constructores.

Así la invocación a un procedimiento $r(a)$ del objeto x ($x.r(a)$), el cual produce un cambio de estado en nuestro sistema, podremos modelarlo como una función de la forma $A \nrightarrow States \nrightarrow States$ donde A denotan los parámetros del procedimiento y la función $States \nrightarrow States$ describe el nuevo estado producido por la ejecución del procedimiento en términos del estado previo.

De esta manera los posibles cambios de estados, dado que un estado está definido como un conjunto de objetos y una colección de funciones sobre esos objetos, son

1. Cambiar una de las funciones en el primer componente del estado; al nivel mas básico esto significa cambiar el valor de una de las funciones en uno de sus posibles argumentos.
2. Cambiar el conjunto de objetos presente en la segunda componente del estado. [2]

2.2. Semántica de los operadores básicos

A continuación presentamos la semántica formal de un subconjunto del lenguaje `Eiffel`. Esto permite "mapear" programas a funciones matemáticas que los modelan. De esta forma se define una semántica funcional de un lenguaje imperativo, permitiendo de esta manera introducir la noción de función de abstracción en este ultimo paradigma, como veremos mas adelante.

2.2.1. El operador de asignación

La operación fundamental en el paradigma imperativo, transversal al orientado a objetos, es la asignación. En nuestro modelo, la asignación, como transformador de estados básico, estará representada por una función denominada "sustitución de funciones" y denotada como ":=", cuyo perfil es $(Objects \nrightarrow States \nrightarrow Objects) \nrightarrow (Objects \nrightarrow States \nrightarrow Values) \nrightarrow Objects \nrightarrow States \nrightarrow States$. Así, se permite una sustitución de la forma

$$f := E$$

De esta manera definimos la semántica del operador de asignación de Eiffel, mediante la función $:=$, del siguiente modo [3]

Regla 1 (asignacion)
$\overline{f := g} \doteq [\overline{f} := \overline{g}]$

El operador de substitución de funciones, intuitivamente, dado un objeto o y un estado s, retorna un estado s' en donde solo cambia f para un único elemento de su dominio como se muestra en la fig. 1.

Para nuestro trabajo utilizaremos las siguientes propiedades básicas de este operador, cuya demostración puede ser encontrada en [Blanco 05] junto con una definición formal de $:=$.

[2] Notar que esto no implica que se modifique el conjunto $Objects$ definido anteriormente.

[3] En adelante para denotar la semántica asociada a un constructor del lenguaje c, utilizaremos la notación $\overline{c}$, y se utilizarán además los corchetes para agrupar funciones

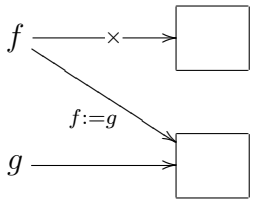

Figura 1: Ilustración de la Asignación

Propiedad 1
$f\ o\ ([f := g]\ o\ s) = g\ o\ s$

Propiedad 2
$o \neq o' \vee f \neq h \Rightarrow h\ o\ ([f := g]\ o'\ s) = h\ o\ s$

2.2.2. Secuenciamiento de instrucciones

La semántica para el secuenciamiento de instrucciones hace uso de un nuevo operador de composición de funciones, denominado *rightmost composition* (se denota ■), y se define como[4]

Definición 1 (rgmc)
$\blacksquare\ ::\ (A \nrightarrow B \nrightarrow C) \nrightarrow (A \nrightarrow C \nrightarrow D) \nrightarrow A \nrightarrow B \nrightarrow D$
$[f \blacksquare g]\ a\ b\ \doteq\ [[f\ a] \square [g\ a]]\ b$

Dado que las instrucciones son abstraídas en funciones, el modelo para el secuenciamiento de las mismas, es la composición de sus modelos. De esta manera se define su semántica como

Regla 2 (secuenciamiento)
$\overline{i; j}\ \doteq\ [\overline{i} \blacksquare \overline{j}]$

Es importante notar que tanto el perfil de i como el de j es de la forma $Objects \nrightarrow States \nrightarrow States$, dado que los mismos son transformadores de estados.

2.2.3. Comandos condicionales

Se define la semántica de los comandos condicionales de manera tradicional utilizando expresiones condicionales del lenguaje funcional

Regla 3 (if)
$\overline{\textbf{if } b \textbf{ then } i \textbf{ else } j \textbf{ end}}\ \doteq\ \mathsf{if}\ \overline{b}\ \mathsf{then}\ \overline{i}\ \mathsf{else}\ \overline{j}$

La expresión condicional if se encuentra además definida de la siguiente manera

Definición 2 (if**)**
$[\mathsf{if}\ \overline{b}\ \mathsf{then}\ \overline{i}\ \mathsf{else}\ \overline{j}\ \mathsf{end}]\ o\ s \doteq \mathsf{if}\ \overline{b}\ o\ s\ \mathsf{then}\ \overline{i}\ o\ s\ \mathsf{else}\ \overline{j}\ o\ s$

dado que el perfil de b es de la forma $Objects \nrightarrow States \nrightarrow Bool$ y de $i,\ j$ de la forma $Objects \nrightarrow States \nrightarrow States$

[4]El operador □ denota la composición de funciones usual

2.2.4. Comandos iterativos

La semántica de los comandos iterativos, se define recursivamente de la siguiente manera

Regla 4 (loop) .
$\overline{\textbf{from } i \textbf{ until } b \textbf{ loop } j \textbf{ end}} \doteq i \blacksquare [\overline{\textbf{until } b \textbf{ loop } j \textbf{ end}}]$

Regla 5 (until) .
$\overline{\textbf{until } b \textbf{ loop } j \textbf{ end}} \doteq \mathsf{until}\ \overline{b}\ \mathsf{loop}\ \overline{j}$

Definición 3 (until**)** . *Definimos a* until *como*
$[\mathsf{until}\ b\ \mathsf{loop}\ j]\ o\ s \doteq \mathsf{if}\ b\ o\ s\ \mathsf{then}\ s\ \mathsf{else}\ [j \blacksquare [\mathsf{until}\ b\ \mathsf{loop}\ j]]\ o\ s$

2.2.5. Llamada a rutinas

La semántica de la llamada a rutinas utiliza una variante de la composición de funciones usual (denotada **.**) y definida como

Definición 4 (.)
$\centerdot :: (A \nrightarrow B \nrightarrow C) \nrightarrow (C \nrightarrow B \nrightarrow D) \nrightarrow (A \nrightarrow B \nrightarrow D)$
$[f \centerdot g]\ a\ b \doteq g\ (\ f\ a\ b\ )\ b$

Cuando se invoca una rutina `f.r(a`$_1$`, ...,a`$_n$`)`, donde `r` tiene parámetros formales `x`$_1$,..., `x`$_n$, antes que `r` comience a ser ejecutada, ocurre que estos últimos son ligados con los parámetros efectivos(o actuales), de modo tal que referencien a los mismos objetos. En el nivel mas básico del sistema formal, el efecto anterior puede ser modelado en términos de nuestro operador de sustitución de funciones a través de la siguiente regla

Regla 6 *Llamadas a rutinas*
Sea `r` *con parámetros formales* `x`$_1$,..., `x`$_n$, *luego*
$\overline{f.r(h_1, ..., h_n)} = [f \centerdot x_1 := h_1] \blacksquare ... \blacksquare [f \centerdot x_n := h_n] \blacksquare f \centerdot r(x_1, ..., x_n)$

Luego puede demostrarse la siguiente propiedad

Propiedad 3
$\overline{f \centerdot set\_g(h)} \blacksquare f \centerdot g \stackrel{*}{=} h$

3. El modelo asociado a una clase

Si queremos especificar que una clase satisface su contrato, el primer paso es la elección de un modelo asociado la misma.

Dado que estas clases (en particular las mas básicas) están destinadas a modelar álgebras abstractas, sus métodos deberán implementar las funciones de esta ultima, y sus instancias (objetos) se corresponderán con elementos pertenecientes al tipo de interés del álgebra abstracta.

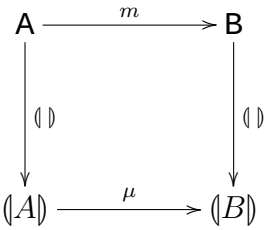

Figura 2: Corrección del método concreto en base a su especificación abstracta

Ahora, al momento de la verificación de un método m que implementa una función abstracta μ, deberemos probar que la aplicación de m sobre un objeto o, tiene el mismo efecto que la aplicación de μ sobre el elemento del tipo de interés que o representa. Es decir, debemos ver que el diagrama de la figura 2 conmuta

Pero entonces se plantea el interrogante de como es posible obtener este último a partir del objeto o. Surge entonces el concepto de modelo.

El modelo asociado a una clase lo representamos a través del atributo $model$, el cual es una función que tiene como rango el tipo de interés del álgebra abstracta que la clase particular representa, y cuyo propósito es definir la función de abstracción para cada una de sus instancias, es decir, aquella función que nos posibilitará obtener el elemento que estos representan.

Así, `model` aparece como un atributo más de la clase, con el propósito de especificar los contratos de la misma y probar su corrección. Su definición estará dada, de acuerdo al formalismo que hemos presentado, en un lenguaje funcional.

Notar que la semántica de `model` nos arrojará una función que tiene como rango un tipo abstracto, que no forma parte del sistema de tipos de `Eiffel`. Sin embargo, a nivel de implementación no habrá problemas puesto que $model$ será usado solo para las aserciones (no apareciendo explícitamente en el cuerpo del programa), y por contrapartida este enfoque nos permitirá escribir post condiciones concisas y expresivas.

De este modo, se tienen ahora herramientas para definir las funciones de abstracción sobre objetos. Para ilustrar sobre la forma en la cuál se puede especificar el modelo asociado a una clase damos a continuación dos ejemplos.

Primero definimos a $model$ para la clase `NODE[G]`, la cuál va a representar una secuencia cuyos elementos serán las abstracciones de los objetos contenidos en el atributo `item` de cada uno de los objetos del tipo `NODE[G]` que se hallen conectados entre si vía su otro atributo `next`.

Declaramos a `NODE` de la siguiente manera:

```
class NODE[G]
  ...
  feature{NONE}
    item: G
    next: NODE[G]
  ...
  feature{SPECIFICATION}
    model :: Objects ⇸ States ⇸ [(|G|)]
```

```
model = abst id

abst:: Objects → Objects ↛ States ↛ [(|G|)]
abst f o s =
if f o s = ⊥
  then []
  else [(|f.item|)] ⋮* (|f.next|)] o s
...
```

Aquí, **id** denota la función identidad, $\overset{*}{:}$ es el constructor de listas[5] que dado un elemento x y una lista xs retorna una lista cuyo primer elemento es x y el resto de la lista es xs. Mientras que, para referirnos al rango de la función $model$ definida para una clase A, usaremos la notación $(|\mathsf{A}|)$. Así por ejemplo $(|\texttt{INTEGER}|) = \mathbb{Z}$, al estilo de la semántica denotacional clásica.

A su vez, el operador $(|\ |)$ se sobrecarga para denotar la función de abstracción sobre objetos, la cual se define formalmente como:

Definición 5
$(|\ |) :: (Objects \to States \to Objects) \to Objects \to States \to A$
$(|f|) = f\,.\,model$

En la definición anterior cabe la posibilidad que la lista sea infinita, en el caso de que un ciclo exista entre los objetos de la clase NODE. La prohibición de esta condición podría constituir el invariante de clase de LINKED_LIST, la cuál puede implementarse utilizando objetos de tipo NODE.

4. Un caso de estudio

Mostramos en esta sección un ejemplo de la aplicación de todas las herramientas teóricas expuestas en el presente trabajo, donde se pondrán de manifiesto todas las carencias existentes a la hora de establecer la corrección de clases en ausencia del formalismo aquí presentado, las cuáles son enumeradas en la conclusión. En adición veremos de que manera las post condiciones de los métodos son de gran ayuda a la hora de modularizar las pruebas.

Dado el siguiente método de la clase LINKED_LIST implementado en Eiffel y anotado con aserciones, probaremos que el cuerpo del ciclo mantiene el invariante[6].

```
in( x: G ): BOOLEAN is
  do
    from
      g := f; b := false
    invariant I1 ⋀* I2
    variant #* (|g|)
    until ( b or g = Void ) loop
      b := ( x.is_equal( g.get_item ) ); g := g.get_next
```

[5]Hay en realidad una sutil diferencia sobre la cual no conviene ahondar en el presente trabajo. Sin embargo esta omisión no presentara mayores obstáculos

[6]Para evitar extendernos demasiado en el trabajo, no se expone la prueba de corrección total

```
    end
    Result := b
ensure  Result ≡* (|x|) in* (|f|)
end
```

donde

$I_1 : b \overset{*}{\equiv} (|x|) \overset{*}{in} ((|f|) \overset{*}{\uparrow} (\overset{*}{\#} (|f|) \overset{*}{-} \overset{*}{\#} (|g|)))$

$I_2 : (|f|) = ((|f|) \overset{*}{\uparrow} (\overset{*}{\#} (|f|) \overset{*}{-} \overset{*}{\#} (|g|))) \overset{*}{+\!\!+} (|g|)$

Antes de comenzar con la prueba hacemos las siguientes aclaraciones con respecto a la notación y funciones usadas:

- Las funciones lógico-matemáticas que tienen un asterisco arriba pueden considerarse como las usuales. Si bien no son las mismas funciones, tienen idéntica semántica, difiriendo solo en los tipos de sus argumentos iniciales[7].
- $x \overset{*}{in} xs$ y retorna verdadero si y solo si el elemento se halla contenido en xs.
- La función $\overset{*}{\#}$ retorna el numero de elementos contenidos en una lista.
- $xs \overset{*}{\uparrow} n$ retorna una secuencia con los primeros n elementos de xs; y $xs \overset{*}{:} n$ retorna el n-ésimo elemento de la secuencia xs [8].
- $\overset{*}{+\!\!+}$, es el operador de concatenación de secuencias.

Así, debemos demostrar, siendo `Body` el cuerpo del ciclo

$$(I_1 \overset{*}{\wedge} I_2) \overset{*}{\wedge} \overset{*}{\neg} B \overset{*}{\Rightarrow} \texttt{Body} \blacksquare I_1 \tag{1}$$

El uso de funciones de abstracción, nos permite hablar de propiedades asociadas a las implementaciones. Así, de acuerdo a las post condiciones de los métodos usados en la implementación de la función `in`, es posible inferir (prueba omitida) las siguientes propiedades[9] que nos servirán en la prueba de corrección

1. $\overline{\texttt{g.get\_next}} \overset{*}{=} (|g|) \overset{*}{\downarrow} \overset{*}{1}$
2. $\overline{\texttt{g.get\_item}} \overset{*}{=} \overset{*}{hd}\ (|g|)$
3. $\overline{\texttt{x.is\_equal(y)}} \overset{*}{\equiv} (|x|) \overset{*}{=} (|g|)$

[7] A los efectos de brevedad no definimos el operador $*$. Sin embargo esto no dificultara la comprensión de los temas aquí tratados

[8] Es claro que estas funciones son parciales, no están definidas en caso que la secuencia posea menos de n elementos

[9] Estas, nos permiten comprender la función que realizan estos métodos de una manera simple, sin necesidad de revisar el código para esto

Demostremos la propiedad 1 enunciada anteriormente

Sea $[b := \overline{\texttt{(x.is\_equal(g.get\_item))}}] \blacksquare [g := \overline{\texttt{g.get\_next}}] \blacksquare I_1$
$\equiv$ { hipotesis }
$[b := (\!|x|\!) \overset{*}{=} (\!|hd\ \ g|\!)] \blacksquare [g := (\!|g|\!) \overset{**}{\downarrow} 1] \blacksquare I_1$
$\equiv$ { def. I_1; prop. $(\!|(\!|x|\!)|\!) \overset{*}{=} (\!|x|\!)$ }
$[b := (\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!)] \blacksquare [g := (\!|g|\!) \overset{**}{\downarrow} 1]$
$\blacksquare b \overset{*}{\equiv} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{\uparrow} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!)))$
$\equiv$ { prop. $Q(x \leftarrow E) \overset{*}{=} [x := E] \blacksquare Q$ }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{\uparrow} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ ((\!|g|\!) \overset{**}{\downarrow} 1)))$
$\equiv$ { prop. $(\!|(\!|x|\!)|\!) \overset{*}{=} (\!|x|\!)$ }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{\uparrow} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!) \overset{**}{\downarrow} 1))$
$\equiv$ { prop. $\#Xs \downarrow 1 = \#Xs - 1$; algebra }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{\uparrow} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!) \overset{**}{\downarrow} 1 + 1))$
$\equiv$ { prop. $Xs \uparrow (n+1) = (Xs \uparrow n) \overset{*}{+\!\!+} (Xs.n)$; hipotesis $(g \overset{*}{\neq} Void)$ }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{\uparrow} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!))\ \overset{*}{+\!\!+}\ (\!|f|\!) \overset{*}{:} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!)))$
$\equiv$ { prop. in }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{\uparrow} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!))) \overset{*}{\vee} (\!|x|\!)\ in\ ((\!|f|\!) \overset{*}{:} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!)))$
$\equiv$ { hipotesis; logica }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ([(\!|f|\!) \overset{*}{:} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!))])$
$\equiv$ { prop. $As \overset{*}{+\!\!+} Bs.(\#As) = hd\ Bs$ }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!)\ in\ ([(\!|f|\!) \overset{*}{:} (\overset{*}{\#}\ (\!|f|\!)) \overset{**}{-\#}\ (\!|g|\!))])$
$\equiv$ { def. in }
$(\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!) \overset{*}{\equiv} (\!|x|\!) \overset{*}{=} hd\ \ (\!|g|\!)$

QED

Así demostramos la implicación planteada en 1.

5. Conclusión

La semántica aquí introducida ha permitido detectar casos de aliasing como el del ejemplo expuesto en la introducción, satisfaciendo así un agudo requerimiento que aparece al razonar con programas que contienen referencias a memoria. Si bien aún se están sentando las bases de este marco de trabajo matemático, los resultados que se consiguieron hasta aquí nos hace mirar el futuro con optimismo.

Una de las carencias a la hora de especificar e implementar tipos abstractos de datos en lenguajes imperativos es que no se dispone de una semántica apropiada para la introducción de funciones de abstracción. Por ello uno de los logros de esta semántica es permitir trabajar con este tipo de funciones dentro del paradigma imperativo.

Como se puede apreciar en el caso de estudio, si no contáramos con las herramientas teóricas que presentamos, no se podría demostrar, mediante formalismos como la lógica de Hoare, que el método implementa correctamente la operación del álgebra de las secuencias. Principalmente porque no tendríamos forma de ir de clases concretas a álgebras abstractas.

La función de abstracción asociada a cada clase permite expresar los contratos de una manera más simple y completa, evitando así subespeficaciones o sobre especificaciones. Por lo que a diferencia de lo que se establece en [Meyer 03b], los contratos de las clases de librerías como `Eiffel Base` no deberían ser extendidos, sino que deberían ser reformulados.

Es importante señalar la utilización de las post condiciones de los métodos utilizados en la prueba de corrección del ejemplo que aquí presentamos, lo que permite modularizar las demostraciones, simplificando las mismas.

5.1. Trabajos futuros

El primer paso hacia la prueba de corrección de clases es extender la la semántica hasta ahora definida al resto de los constructores del lenguaje `Eiffel`.

Luego, es menester incorporar la semántica dentro de algún demostrador de teoremas de manera de generar automáticamente las obligaciones de prueba.

Por otro lado es nuestra meta extender el proceso realizado en el presente trabajo a una amplia variedad de estructuras básicas y algoritmos fundamentales. Esto involucra construir y desarrollar las teorías asociadas a estas clases, extender los contratos de las mismas, realizar las pruebas y refinar las mismas en el proceso.

Todos estos trabajos futuros conforman una linea importante de investigación, que se inserta dentro del creciente interés de la comunidad de orientación a objetos en los métodos formales.

Referencias

[EWD 76] E. W. Dijkstra. *A discipline of programming*, Prentice-Hall 1976.

[EWD 1036] E. W. Dijkstra. *On the cruelty of really teaching computer science, circulated privately*, 1988.

[Meyer 03b] B. Meyer. *A Framework for Proving Contract-Equipped Classes*, In *Abstract State Machines 2003 Advances in Theory and Aplications*, Springer-Verlag 2003.

[Meyer 03a] B.Meyer. *Towards Practical Proofs of Class Correctness*, In *ZB 2003: Formal Specification and Developement in Z and B, pages 359-387*, 2003.

[Meyer 97] B.Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice Hall 1997.

[Blanco 05] J.Blanco, P.Castro. *Una Semántica para demostrar Correcion de Clases*, Reporte Tecnico, Universidad Nacional de Río Cuarto 2005