Menos es Más

Una formalización minimalista de la Orientación a Objetos

Trabajo de Grado para la Licenciatura en Informática
Facultad de Ciencias Exactas
Universidad Nacional de La Plata

Director:	**Gabriel Baum**
Codirector:	**Máximo Prieto**
Alumnos:	**Verónica Argañaraz**
	María José Presso
	Natalia Romero

Indice

Capítulo 1

Introducción

1. Introducción

La orientación a objetos es un acercamiento al desarrollo de software que permite construir sistemas en forma modular, provee un marco general para las distintas etapas del desarrollo, y facilita la extensión y el reuso.

El paradigma de orientación a objetos presenta características que le son propias y que lo distinguen de los demás paradigmas existentes. Sin embargo, no existe un consenso acerca de cuales de éstas son las esenciales del paradigma, y por esto los distintos lenguajes presentan características muy diferentes. Una de las características más importantes, presente en todos los lenguajes orientados a objetos, es la capacidad de compartir comportamiento, conocida como *sharing*, que permite que un objeto exhiba comportamiento que no define por sí mismo, sino que está definido por algún otro objeto. Hay dos organizaciones clásicas para compartir comportamiento, que dan lugar a dos diferentes modelos y familias de lenguajes Orientados a Objetos.

Uno de estos modelos se basa en el concepto de clase. En este modelo, las clases definen comportamiento para una colección de objetos creados a partir de ella, llamados instancias de la clase. Todo objeto pertenece a una clase, y obtiene de ella los métodos que necesita para responder a los mensajes que recibe.

El otro modelo se basa en el concepto de prototipo. En este modelo, un objeto puede ser creado por otro objeto (el prototipo), y puede tomar comportamiento de cualquier otro objeto.

Se discutió largamente cuál de estos dos modelos es más apropiado para el desarrollo de aplicaciones, y cuál es el más básico, en el sentido de capturar la esencia del paradigma. [Lie86] muestra como implementar mecanismos para compartir comportamiento entre grupos de objetos usando prototipos. [Ste87] muestra como simular prototipos usando clases.

En cuanto al desarrollo de aplicaciones, cada modelo es más apropiado para diferentes dominios y etapas. Los prototipos son muy flexibles y permiten prototipación rápida y programación exploratoria. Son útiles para dominios poco conocidos, que se están investigando. Las clases permiten una buena estructuración de los sistemas, y son útiles cuando se comprende bien el dominio y para construir componentes reusables. Una aplicación evolucionaría desde el modelo de prototipos en la etapa inicial, de exploración, hacia un modelo de clases en su madurez. [SLU88]

Para comprender mejor el paradigma, es importante contar con una formalización de los conceptos del mismo, definiéndolos rigurosamente de manera de poder razonar acerca de ellos. En este trabajo se busca construir un modelo formal del paradigma de Orientación a Objetos que esté basado en una formulación minimal del mismo.

El modelo deberá comprender otras nociones importantes del paradigma representadas a partir de las nociones más elementales. Una de estas nociones es la capacidad de compartir comportamiento, tanto mediante clases como mediante prototipos.

El modelo buscado deberá cumplir con determinadas características:

- ser formal
- ser suficientemente expresivo para poder representar los conceptos del paradigma
- ser adecuado en relación al paradigma de Orientación a Objetos

- ser lo suficientemente simple como para que el conocimiento lógico o matemático necesario para entender o expresar los conceptos básicos de objetos sea mínimo.

La construcción de los demás conceptos usando sólo las nociones mínimas permite integrarlos en un marco único. Esto es interesante no sólo desde el punto de vista teórico, sino también desde el práctico. En el caso de clases y prototipos esto provee un espacio en el cual un sistema puede evolucionar de un modelo al otro a medida que progresa su desarrollo.

Los primeros trabajos en formalización de los conceptos de orientación a objetos se concentraron en definir la semántica de la herencia entre clases, y consideraron a las clases como elementos primitivos. Más recientemente se desarrollaron otros formalismos que tienen como primitivos a los objetos y no a las clases.

En este trabajo analizamos los conceptos presentes en el paradigma de Orientación a Objetos, identificando un conjunto de conceptos esenciales, y luego clasificándolos entre básicos o elementales y construibles.

Los elementos básicos son los que estarán presentes en la formulación minimal. Esta formulación minimal es un cálculo de objetos imperativo con extensiones. Tiene primitivas de creación de objetos, pasaje de mensajes y actualización y extensión de objetos. Este cálculo es una variante del cálculo imperativo de objetos de Abadi y Cardelli [AC96].

Dentro de los conceptos que consideramos construibles ubicamos a la capacidad de compartir comportamiento, o sharing. Esta característica es una de las más importante, ya que se encuentra presente en casi todos los lenguajes, y su organización define las dos grandes familias de lenguajes y los dos modelos tradicionales del paradigma. También es importante desde el punto de vista de la ingeniería de software, porque permite una modelización más adecuada de la realidad, favorece el reuso de software, evita la replicación de código y simplifica la definición de políticas de seguridad.

Las diferentes formas en que se presenta el sharing en los distintos lenguajes se pueden ver como combinaciones de decisiones más simples. Presentaremos una clasificación que permite describir los distintos esquemas de sharing posibles.

El sharing se puede construir en base a los elementos provistos en la formulación minimal. [AC96] y [FM98] muestran como construir clases, es decir, sharing por grupos de objetos, en cálculos similares al nuestro.

En este trabajo se muestra como representar mecanismos de sharing por objeto, en un formalismo que no tiene primitivas para compartir comportamiento, basado solamente en objetos y mensajes. Según nuestro conocimiento, esta es la primera construcción con estas características [ABPPPR97, APR98].

La organización del trabajo es la siguiente: en el próximo capítulo se presentarán los conceptos generales de orientación a objetos, según aparecen en la literatura. En el capítulo 3 se definen los conceptos que tomamos como esenciales para la construcción del modelo, y se discute especialmente el sharing. En el capítulo 4 se presenta el cálculo imperativo de objetos de Abadi y Cardelli, y la forma de representa sharing por grupo en ese cálculo. En el capítulo 5 se presenta el cálculo imperativo de objetos con extensiones. En el capítulo 5 se define Asterix, un lenguaje de alto nivel con primitivas para las distintas formas de sharing por objeto. En el capítulo 7 se muestra como se traducen las construcciones de Asterix al cálculo imperativo con extensiones. En el capitulo 8 se discute la relación con otros trabajos y finalmente en el capítulo 9 se comentan las conclusiones del trabajo. En el apéndice se encuentra la definición completa de la función de traducción de Asterix al cálculo.

Capítulo 2

Conceptos de Orientación a Objetos

2. Conceptos de Orientación a Objetos

En esta sección recopilaremos los conceptos y términos que aparecen en la literatura como característicos de la Orientación a Objetos. Luego en el capítulo 3 realizaremos una síntesis de los mismos, identificando y describiendo aquellas características que consideramos fundamentales dentro del paradigma.

Clases

Los programas orientados a objetos usualmente operan con cientos o miles de objetos. Muchos de estos objetos tienen propiedades comunes, son de la misma especie. Estas propiedades en común no necesitan ser descriptas separadamente para todos estos objetos individuales. En lugar de esto, podemos definir una clase de objetos y describir los objetos de esta clase, qué mensajes deberían aceptar y qué métodos deberían ejecutar en respuesta a estos mensajes. Los objetos y las clases están tan cercanamente acoplados que muchos programadores creen que la programación orientada a objetos sin clases no sería posible [Blas94]. [Mey97] tiene esta visión, afirmando que "El método orientado a objetos está basado en el concepto de clase. Informalmente, una clase es un elemento de software que describe un tipo abstracto de datos y su implementación parcial o total. Un tipo abstracto de datos es un conjunto de objetos definidos por la lista de operaciones o características ('features') aplicable a estos objetos, y las propiedades de estas operaciones. El método y el lenguaje (de desarrollo orientado a objetos) deberían tener la noción de clase como su concepto central."

Smalltalk es un lenguaje basado en el concepto de clase. En Smalltalk, " una clase describe la implementación de un conjunto de objetos que representan todos la mima especie de componentes de un sistema. Los objetos individuales descriptos por una clase se llaman sus instancias. Una clase describe la forma de las memorias privadas de sus instancias y cómo llevar a cabo sus operaciones. Todo objeto en el sistema Smalltalk-80 es una instancia de una clase. Aún un objeto que representa un componente del sistema único se implementa como la única instancia de una clase. Programar en el sistema Smalltalk-80 consiste en crear clases, crear instancias de clases y especificar secuencias de intercambios de mensajes entre estos objetos." [Gold89]

"Las instancias de una clase son similares tanto en sus propiedades públicas como privadas. Las propiedades públicas de un objeto son los mensajes que componen su interface. Todas las instancias de una clase tienen la misma interface de mensajes ya que representan la misma especie de componentes. Las propiedades privadas de un objeto son un conjunto de *variables de instancia* que componen su memoria privada y un conjunto de métodos que describen cómo realizar sus operaciones. Las variables de instancia y los métodos no son directamente accesibles a otros objetos. Todas las instancias de una clases usan el mismo conjunto de mensajes para describir sus operaciones."[Gold89]

"Las clases tienen diferentes roles en la programación orientada a objetos. Primero, podemos verlas como un medio de identificar objetos con las mismas propiedades. En este sentido, una clase es un concepto abstracto que simplemente nos ayuda a distinguir objetos con diferente estructura y comportamiento. Segundo, las clases también pueden ser vistas como un mecanismo de estructuración, similar a un procedimiento o un módulo. Este aspecto es particularmente importante para la legibilidad y mantenibilidad de los programas porque todos los aspectos de una cierta especie de objetos están descriptos en un solo lugar del programa. Esta localidad es esencial para el ocultamiento de información y también ayuda a rastrear

errores. Tercero, una clase también se usa como un medio para crear objetos en muchos lenguajes de programación. En este sentido, una clase es considerada una 'fabrica de objetos', ella 'sabe' cómo crear objetos con la estructura y el comportamiento que ella define"[Blas94]

Los lenguajes basados en clases permiten construir clases derivando una nueva clase de una existente. Esto permite representar objetos que son sustancialmente similares, pero tienen alguna característica particular. La nueva clase tiene inicialmente todas las propiedades de la clase en la cual se basa. Se pueden agregar propiedades (variables de instancia, mensajes y métodos) a la nueva clase, y cambiar algunas propiedades existentes para adaptarlas a las necesidades de la nueva clase. Las propiedades que no se definen en la nueva clase, se toman automáticamente de la original. Esta propagación de propiedades se llama *herencia*. La nueva clase se llama *subclase* de la original, y ésta se llama *superclase* de la nueva.[Blas94]

Objetos y mensajes

El paradigma de orientación a objetos, como aproximación a la construcción de software, abarca todo el ciclo del desarrollo [Mey97]. Se basa en una correspondencia intuitiva entre la simulación por medio de software de un sistema del mundo real, y el mundo real en sí mismo. Por analogía a los objetos del mundo real, los componentes de software se llaman *objetos*.

Esta correspondencia tiene propiedades que favorecen el desarrollo de software, integrando buenas técnicas de análisis, diseño e implementación en un marco intuitivo y uniforme. La analogía entre los modelos de software y el mundo real facilita la etapa de análisis. La construcción de abstracciones contribuye a la elasticidad de los modelos, permitiendo la evolución del diseño. La construcción de taxonomías durante el análisis, diseño e implementación favorece el reuso. [AC96]

[Blas94] hace una presentación general de la orientación a objetos. Para ello, comienza analizando las definiciones de objeto dadas por el diccionario Webster: "cualquier cosa visible o tangible; un producto o substancia material", "visión; apariencia; representación", también parafrasea la acepción de objeto en filosofía: "en programación, cualquier cosa que puede ser conocida o percibida por la mente del programador".

Los objetos representan estado, y también pueden realizar acciones. En este sentido, resume la esencia de qué es un objeto con la ecuación *objeto = estado + comportamiento*

[Blas94] compara un programa orientado a objetos con el motor de un auto: un conjunto de partes relativamente simples, cuya combinación e interacción resulta en una construcción compleja. No hay un algoritmo que controle el trabajo del motor, sino que cada parte debe contribuir para que todo el motor funcione. Otro parecido es el hecho de que las partes individuales ordenan a otras realizar ciertas acciones. Una propiedad de esto es que las partes conectadas no dependen de aquellas a las que están conectadas, sólo se requiere que dos partes que se comunican tengan una interface común. Esto significa que las partes pueden ser intercambiadas dentro de ciertos límites sin inhabilitar el motor completo. Similarmente, en la programación orientada a objetos, los objetos tienen estado, interfaces y comportamiento. Los objetos pueden pensarse como datos con algoritmos incorporados. Según [Gold89], "un objeto consiste de una memoria privada y un conjunto de operaciones".

Los objetos se comunican entre sí por medio de mensajes. Un mensaje es una operación abstracta. Un mensaje es un pedido a un objeto de realizar una de sus operaciones. La invocación de un mensaje se llama envío de mensaje. El objeto al que se le envía el mensaje es el *receptor* y es quien determina como llevar a cabo la operación requerida. El mensaje no especifica cómo se realiza la operación y en general los clientes no lo saben (y no deberían

saberlo). La computación se ve como una capacidad intrínseca de los objetos que puede ser invocada uniformemente enviando mensajes [Blas94, Gold89].

El conjunto de mensajes a los que un objeto puede responder se llama su *interface* con el resto del sistema[Gold89] o también su *protocolo* [Blas94]. "La única forma de interactuar con un objeto es a través de su interface. Una propiedad crucial de un objeto es que su memoria privada puede ser manipulada solamente por sus propias operaciones. Una propiedad crucial de los mensajes es que son la única forma de invocar las operaciones de un objeto. Estas propiedades aseguran que la implementación de un objeto no puede depender de los detalles internos de otros objetos, sólo de los mensajes a los cuales éstos responden. Los mensajes aseguran la modularidad del sistema porque especifican el tipo de operación deseada, pero no cómo debe ser efectuada."[Gold89].

"El emisor del mensaje debería saber solamente cómo solicitar una acción particular del receptor, pero nunca cómo el receptor efectivamente la realizará. De manera similar, el receptor nunca debería necesitar conocer la identidad del emisor ni el contexto en el cuál se llevará a cabo. Estas reglas estrictas garantizan que tanto el emisor como el receptor pueden ser reemplazados con otros objetos y que los objetos pueden ser usados en diferentes entornos. El *ocultamiento de información* es por lo tanto un concepto clave de la programación orientada a objetos que asegura la reusabilidad de los objetos" [Blas94]. Según [Mey97] el ocultamiento de información es el mecanismo que hace que algunas características sean inaccesibles para las invocaciones de los clientes. Esto es esencial para la evolución suave de los sistemas de software. Dice además que no es suficiente que el mecanismo de ocultamiento de información soporte características exportadas (disponibles para todos los clientes) y características secretas (disponibles para ningún cliente); debería ser posible también exportar una característica selectivamente a un conjunto de clientes especificados.

Cuando un objeto recibe un mensaje, reacciona ejecutando un conjunto de sentencias, un algoritmo desarrollado por el programador. Este algoritmo se llama método [Blas94]. Cada método describe cómo llevar a cabo la operación requerida por un tipo particular de mensaje. Un método puede especificar algunos cambios en la memoria privada, y/o enviar algunos otros mensajes. Un método también especifica un objeto que debe ser retornado como el valor del mensaje invocado. Los métodos de un objeto pueden acceder al estado interno del propio objeto, pero no aquellos de otros objetos [Gold89]. Pueden existir, y usualmente hay métodos con el mismo nombre en distintos objetos. Por lo tanto, no es posible decir que método se ejecutará para un envío de mensaje determinado. El método apropiado se seleccionará en tiempo de ejecución, de acuerdo al receptor del mensaje. Este proceso se conoce como *binding dinámico* [Mey97]. El hecho de poder enviar el mismo mensaje a distintos objetos, y que cada uno pueda reaccionar de manera diferente se denomina *polimorfismo*.

Los objetos tienen identidad. Cada objeto creado en un sistema orientado a objetos tiene una identidad única, independiente de los valores de sus propiedades. Dos objetos con diferentes identidades pueden tener idénticas propiedades. Las propiedades de un objeto pueden cambiar, pero esto no afecta la identidad del objeto [Mey97].

Prototipos

Los prototipos son un mecanismo alternativo al de las clases para la construcción de objetos. "La estructura y el comportamiento de objetos similares no está definido en la descripción de una clase, sino por la construcción (generalmente interactiva) de un prototipo. Cada prototipo puede tener su propia estructura y su conjunto de métodos, definiendo así el protocolo del prototipo. Los nuevos objetos se crean simplemente haciendo una copia del prototipo. Por lo tanto, los prototipos no son sólo un concepto abstracto para la definición del comportamiento de

los objetos, sino también un medio de instanciación de nuevos objetos. (...) Un prototipo es un objeto prefabricado con estructura predefinida, contenidos y comportamiento, a partir del cual su pueden crear nuevo objetos copiando el prototipo." [Blas94]

Los lenguajes basados en prototipos se diferencian entre sí por la forma en que los objetos comparten propiedades y en cómo se crean objetos con diferentes propiedades. Según [Lieberman87], "un prototipo representa el comportamiento por defecto para un concepto, y nuevos objetos pueden reusar parte del conocimiento almacenado en el prototipo diciendo de qué manera el nuevo objeto se diferencia del prototipo. El acercamiento de prototipos parece tener algunas ventajas (con respecto al de clases) para representar comportamiento por defecto, y modificar conceptos incrementalmente y dinámicamente. La *delegación* es el mecanismo para implementar esto en los lenguajes orientados a objetos. Después de verificar su comportamiento idiosincrático, un objeto puede reenviar un mensaje a prototipos para invocar un conocimiento más general. (....) cualquier objeto puede ser usado como un prototipo, y cualquier mensaje puede ser reenviado en cualquier momento..."

SELF es un lenguaje basado en el concepto de prototipos. En SELF, un objeto se crea *clonando* (copiando) un prototipo y cualquier objeto puede ser clonado. En SELF todo es un objeto. Los objetos contienen slots con nombre que pueden almacenar tanto estado como comportamiento (es decir, variables o métodos). La herencia en SELF se implementa mediante un slot llamado *parent*. Si un objeto recibe un mensaje y no tiene un slot correspondiente, la búsqueda continua por la referencia al parent. [US91].

Omega [Blas94] es otro lenguaje basado en prototipos. Omega prohibe la delegación y los cambios individuales en los objetos, adhiriendo a un modelo de herencia más tradicional, para permitir localizar las descripciones de los objetos. Un prototipo de Omega no se usa solamente para la instanciación de nuevos objetos, sino que también sirve como un representante de todos los objetos creados directa o indirectamente a partir de él. Por lo tanto, un prototipo representa una clase de objetos con estructura y comportamiento común. Para garantizar que estos objetos tendrán siempre las misma propiedades, todos los cambios relevantes del prototipo se propagan a todos los clones derivados a partir de él. Sólo los contenidos de los objetos son atributos individuales que pueden ser modificados independientemente. Omega permite generar un nuevo prototipo, derivado de uno existente, que en principio tiene las mismas propiedades que el original, pero luego pueden agregarse nuevas propiedades y redefinir las existentes.

Clases vs Prototipos

Según [Lie86] los dos mecanismos para compartir comportamiento en los lenguajes orientados a objetos, clases y prototipos, provienen de distintos enfoques acerca de cómo representar conceptos.

En el primer punto de vista, al cual corresponde el mecanismo de clases, a partir del conocimiento de un cierto elemento se puede construir el concepto de conjunto o clase de ese tipo de elementos, enumerando todas las propiedades esenciales de esos elementos. Para responder acerca de un elemento dado del conjunto, nos referimos a la descripción del conjunto. Si algunos elementos del conjunto comparten ciertas propiedades entre sí, pero no con todos los elementos del conjunto, esta situación se describe mediante una subclase, que comparte todas las propiedades del conjunto original, agregando las propiedades adicionales del subconjunto considerado.

En el segundo punto de vista, se considera al elemento conocido como el elemento prototípico. Si se relaciona un nuevo elemento al anterior, designando a éste como su prototipo, se compartirá conocimiento entre ambos. Cualquier pregunta acerca del nuevo se podrá responder

con la información acerca del original, excepto que exista evidencia en contra de ello, en cuyo caso la información referida al nuevo elemento deberá almacenarse en la descripción de éste último.

Lieberman considera que es más fácil considerar primero ejemplos específicos y luego generalizarlos diciendo que aspectos del concepto pueden variar, y que los prototipos son mejores para expresar conocimiento por defecto.

[Lie86] dice que el mecanismo de delegación es más poderoso que el de herencia. Para representar herencia usando delegación se pueden crear objetos especiales clase que responden a mensajes para crear nuevas instancias. Los objetos clase deben copiar las variables en la cadena de superclases para crear instancias. Los objetos instancias tienen comportamiento para implementar la búsqueda de variables y métodos. Afirma además que la delegación no puede ser implementada mediante herencia.

[Ste87] muestra que herencia y delegación pueden ser usadas para modelar una a la otra. La prueba de que herencia puede modelar delegación se basa en la observación de que la relación de herencia entre clases es delegación, por lo tanto, en lugar de representar los objetos prototipos como instancias, los representa como clases en una jerarquía de herencia sin instancias, capturando la funcionalidad de una jerarquía de delegación.

En [SLU88] los autores de distintas soluciones al problema de representar comportamiento compartido (alternativas al modelo de clases y herencia) presentan una visión integrada de las distintas soluciones. Dicen que una parte fundamental de la naturalidad de expresión provista por la programación orientada a objetos es la habilidad de compartir datos, código y definición, y que con este fin todos los lenguajes orientados a objetos proveen alguna manera de definir un nuevo objeto en términos de uno existente, tomando implementación y descripción de comportamiento del objeto anteriormente definido. Reconocen dos mecanismos fundamentales, templates y empatía, que se pueden usar para analizar y comparar la multitud de mecanismos para compartir propiedades provistos por los distintos lenguajes orientados a objetos. Estos últimos pueden verse como decisiones de diseño en cuanto a la forma que tomará cada uno de los mecanismos fundamentales. En la sección 2 del próximo capítulo analizaremos estos mecanismos en profundidad.

[SLU88] afirman también que no se puede dar una respuesta definitiva acerca de cuál de las elecciones es mejor, sino que diferentes situaciones de programación requieren diferentes combinaciones de estas características: más flexibles para entornos exploratorios y experimentales, y más estrictas para la producción de software en gran escala, relativamente rutinario. Un sistema evoluciona naturalmente de desorganizado y dinámico hacia estático y más optimizado, y por lo tanto la representación de los objetos debería tener también un camino natural de evolución; y los entornos de desarrollo deberían proveer representaciones más flexibles, junto con herramientas, idealmente automáticas, para estructurar el sistema a medida que el diseño se estabiliza.

Capítulo 3

Conceptos Esenciales

3. Conceptos Esenciales

Hasta el momento no existe acuerdo unánime acerca de cuáles son los elementos básicos en el paradigma de Orientación a Objetos. Como se vio en el capítulo anterior, los lenguajes existentes proveen diferentes construcciones para representar distintas características del paradigma. Una forma de lograr una mayor comprensión de los fundamentos del paradigma es establecer cuáles son sus elementos más básicos primitivos y a partir de ellos interpretar los demás conceptos, así como también las construcciones provistas por los distintos lenguajes de programación. Comenzamos este capítulo describiendo todas las nociones que consideramos esenciales.

Según nuestra concepción del paradigma, un *programa* orientado a objetos es un conjunto de *objetos* que se envían *mensajes*. Objetos y mensajes son las únicas nociones esenciales que aparecen en este nivel. Por eso la idea de buscar una formulación minimal del paradigma que se base sólo en esos dos conceptos.

Los *objetos* son abstracciones de entes del mundo real. Un objeto puede representar una abstracción de un ente conceptual, por ejemplo una transacción o una cuenta bancaria, o de un ente físico, como un cliente o un cajero. Que los objetos sean abstraciones quiere decir que representan la esencia del ente modelado. La esencia está representada por la utilidad que la entidad tiene, por su comportamiento.

La manera de responder a los *mensajes* que un objeto recibe es a través de *métodos*. Un método describe un conjunto de *colaboraciones* con otros objetos. Una colaboración es un envío de mensaje. Esos otros objetos que cooperan para que el objeto pueda cumplir con su responsabilidad se llaman *colaboradores*. Podemos distinguir dos tipos de colaboradores: los habituales y los no habituales. Los colaboradores habituales o internos son los que el objeto mantiene siempre para que cooperen con él. En un esquema de clases, los colaboradores habituales son las variables de instancia. Los colaboradores no habituales o externos son aquellos que cooperan ocasionalmente con el objeto. Los objetos pasados como argumentos de mensajes son colaboradores no habituales del objeto que recibe el mensaje.

Una característica que tienen las colaboraciones de un objeto es que son "ciegas", en el sentido de que el objeto no sabe quiénes son sus colaboradores, sólo conoce un nombre para referenciarlos y enviarles mensajes. Esto permite que los colaboradores del objeto puedan ser cambiados sin que él lo perciba, lo único que se requiere es que respondan a los mismos mensajes. Así es como aparece la noción de polimorfismo en las colaboraciones.

Un mecanismo esencial dentro del paradigma es la *creación* de objetos. La manera más básica de crear objetos es a partir de la nada, es decir definiendo su comportamiento desde cero. Otra forma de creación objetos es a partir de otros objetos. En un esquema de clases por ejemplo, existen objetos especializados en crear otros objetos, que son las clases. Las clases contienen la especificación de la esencia (comportamiento) de un cierto tipo de objetos. En otros esquemas existen objetos prototípicos que son clonados para generar nuevos objetos.

La creación de objetos nuevos a partir de otros ya existentes es una manera de *compartir* comportamiento. En general, se comparte comportamiento cada vez que se reusa todo o parte del comportamiento de otro objeto.

Para construir un modelo formal del paradigma de orientación a objetos es necesario precisar más esta interpretación de los que nosotros consideramos elementos esenciales. En la primera sección de este capítulo definimos más rigurosamente esos elementos, y presentamos una clasificación que los agrupa en elementos y mecanismos básicos, y define cuáles son sus

características de comportamiento fundamentales. En la segunda sección analizamos en más profundidad la capacidad para compartir comportamiento, buscando una caracterización que explique y abarque todas sus variantes.

3.1 Elementos y mecanismos básicos

Siguiendo la perspectiva que planteamos al comienzo de este capítulo, podemos caracterizar al paradigma como un conjunto de ***elementos básicos*** que operan mediante un conjunto de ***mecanismos básicos***, y esta forma de operar reúne ciertas ***características*** que, según nuestro análisis, son propias del paradigma. Los ***elementos básicos*** son los **objetos** y los **mensajes**; los ***mecanismos básicos*** son **comunicación** entre objetos, **creación** de objetos y **modificación** de objetos y las ***características*** son **identidad, polimorfismo, encapsulamiento** y **capacidad de compartir** (o sharing). A continuación presentamos una definición más precisa de cada uno de ellos, que usaremos en el resto del trabajo.

Los **objetos, mensajes** y el **mecanismo de comunicación** están íntimamente relacionados, y se explican conjuntamente. Los objetos son las entidades computacionales usadas para modelizar entidades del mundo real. La capacidad fundamental de los objetos es la de comunicarse. La comunicación entre objetos se realiza a través del envío y recepción de mensajes. Un objeto envía un mensaje a otro para notificarle de un evento o para pedirle que realice una tarea. Los objetos son capaces de actuar en respuesta a los mensajes que reciben. La manera de responder a los mensajes es a su vez a través de la comunicación con otros objetos.

Cada objeto tiene un conjunto de mensajes que conoce, que llamamos **protocolo** del objeto. El objeto sabe como responder a los mensajes de su protocolo, decimos que entiende esos mensajes. Existen además mensajes que el objeto no conoce. El objeto también sabe que hacer frente a la recepción de un mensaje que no pertenece a su protocolo.

A cada mensaje que un objeto entiende se asocia un **método**, que es la descripción de la forma de responder al mensaje, es decir, las colaboraciones con otros objetos necesarias para llevar a cabo la tarea encomendada por el mensaje. Algunos de esos objetos que colaboran con el receptor pueden viajar junto con el mensaje, son los **argumentos** del mensaje. La respuesta a un mensaje es a su vez otro objeto, que es devuelto a quien envió el mensaje. Es importante destacar que los métodos para responder a los mensajes también son objetos.

El conjunto de mensajes que un objeto entiende, junto con la forma de responder a esos mensajes conforman el **comportamiento** del objeto. En este contexto, la noción usual de variable de instancia no es necesaria. Una variable de instancia es un mensaje cuyo método asociado representa el valor de la variable.

Además de la comunicación entre objetos, son necesarios los mecanismos de creación y modificación de objetos: la **creación** permite introducir nuevos objetos en el mundo y la **modificación** permite cambiar las comunicaciones que un objeto tiene con otros objetos. La forma más básica de creación de objetos es describiendo en forma explícita todo su comportamiento. Otra forma es definiendo su comportamiento a partir del comportamiento de otros objetos.

Modificar un objeto es modificar su comportamiento. Un mecanismo de modificación de objetos debe permitir cambiar la respuesta a un mensaje, así como también agregar nuevos mensajes al protocolo del objeto. Este ultimo tipo de modificación se llama **extensión** de objetos. Un objeto se extiende incorporando a su protocolo un mensaje que antes no existía y describiendo la respuesta asociada al nuevo mensaje.

La manipulación de los elementos básicos (objetos y mensajes) con los mecanismos básicos (creación, comunicación y modificación) debe exhibir las características básicas es decir, identidad, polimorfismo, encapsulamiento y posibilidad de compartir.

Los objetos tienen una **identidad**, que se mantiene durante toda la vida del objeto, aún cuando el objeto cambie. Así distintos objetos son diferenciables sin importar su similaridad en cuanto a comportamiento, protocolo, etc.

Polimorfismo es la posibilidad de que distintos objetos entiendan el mismo mensaje, y cada uno defina su propia forma de responderlo. Así un mismo mensaje puede enviarse a distintos objetos, y la reacción dependerá del receptor.

Encapsulamiento es una restricción que establece que un objeto solo puede ser modificado por sí mismo, en respuesta a un mensaje. Al restringir la modificación directa desde el exterior, se confiere a los objetos de control autónomo sobre su comportamiento, ya que es el propio objeto quien decide cómo alterarse. Otra restricción de encapsulamiento es que los mensajes que un objeto puede recibir de los otros puedan restringirse a un subconjunto de sus mensajes. Así, el protocolo del objeto puede dividirse en protocolo público –formado por los mensajes que puede recibir desde el exterior- y protocolo privado –formado por los mensajes que sólo el propio objeto puede enviarse.

Sharing es otra característica fundamental de los objetos. Como es un punto que nos interesa especialmente para este trabajo, lo analizamos con más detalle en la siguiente sección.

3.2 Sharing of behaviour

Sharing es la característica que permite a los objetos exhibir comportamiento en común. Es la capacidad por la cual los mensajes (y la forma de responder a ellos) que un objeto define pueden ser reusados por otro objeto. Así, un objeto exhibe comportamiento que no define por sí mismo, sino que está definido por otro objeto.

Uno de los problemas más importantes que hay que resolver al representar sharing, es el que llamamos "manejo del self". Los métodos de un objeto pueden contener referencias al propio objeto; cada vez que el objeto necesita referenciarse a sí mismo utiliza el nombre self. El problema consiste en que cuando un objeto reusa un método definido por otro objeto, las referencias a self que aparecen en ese método deben ligarse al receptor original del mensaje. Es decir que si un objeto *o* toma prestado de otro objeto *p* el método para responder al mensaje *m*, cada vez que *o* reciba el mensaje *m* deberá ejecutarse el método de manera que todas las referencias a self que contenga se liguen a *o*, y no a *p*. Para ello es necesario que las referencias a self no se resuelvan estáticamente, sino en el momento de responder a un mensaje (o ejecución del método).

Para analizar el sharing, nosotros estudiamos las diferentes representaciones existentes buscando identificar los conceptos subyacentes. El objetivo fue encontrar un conjunto de nociones más simples a partir de las cuales se puedan interpretar las construcciones provistas por los lenguajes de programación. Definimos una caracterización de sharing que propone un conjunto de cuatro aspectos básicos que es indispensable tener en cuenta para analizar cualquier mecanismo de sharing.

En la literatura encontramos dos trabajos importantes que proponen clasificaciones de los distintos mecanismos de sharing, el Tratado de Orlando [SLU88] y [AC96].

En el Tratado de Orlando, Stein, Lieberman y Ungar identifican dos mecanismos fundamentales para definir sharing: templates y empathy. Ambos son fundamentales en el sentido de que

ninguno puede definirse en términos del otro y que la mayoría de los lenguajes orientados a objetos pueden describirse de acuerdo a la forma en que combinan los dos mecanismos.

El mecanismo de templates es el que se utiliza para compartir estructura. Un template es un molde que permite crear objetos de su mismo tipo, es decir con las mimas variables de instancia, métodos, punteros a parents, etc. El mecanismo de template garantiza la uniformidad de los miembros de un grupo. En algunos lenguajes los templates son objetos, y en otros están como parte de otros objetos generadores, usualmente llamados clases. Los templates pueden ser estrictos o no estrictos. Un mecanismo de templates estricto no permite que las instancias creadas puedan agregar ni quitar atributos, manteniendo de esta manera exactamente la misma estructura que su template y que las demás instancias creadas a partir de él. Los templates no estrictos relajan en diferentes grados esta restricción, permitiendo que las instancias puedan extenderse, o extenderse y eliminar atributos.

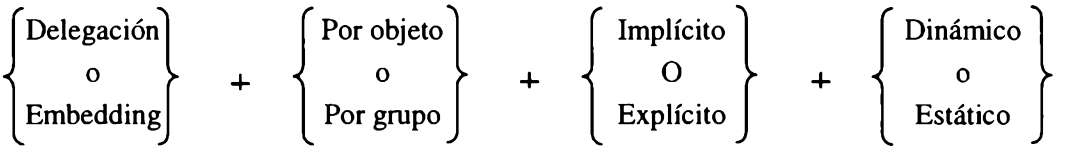
El mecanismo de empathy es lo que permite que un objeto actúe como si fuera otro, permitiendo así compartir comportamiento y estado. Estipulan que un objeto A "empatiza" con otro B en un mensaje determinado, si no tiene manera de responderlo y reacciona como si estuviera tomando prestado el comportamiento de B.

Todas las implementaciones de modelos de herencia y delegación incluyen empathy. Las diferencias están en las decisiones de diseño que tomen respecto a tres dimensiones independientes. La primera dice si el mecanismo es estático o dinámico, es decir si se fija el patrón de sharing cuando el objeto se crea o cuando efectivamente recibe un mensaje. La segunda distingue si es explícito o implícito, según si se permite al programador manejar los patrones de sharing explícitamente o el sistema lo hace en forma automática y uniforme para todos los objetos. La tercera dimensión dice si es por grupo o por objeto, dependiendo de si el comportamiento se define para grupos de objetos o si se puede definir para objetos individuales.

En [AC96] Abadi y Cardelli analizan las características presentes en los diferentes lenguajes basados en clases y basados en prototipos. Al analizar los lenguajes basados en prototipos definen dos aspectos ortogonales en la forma de compartir comportamiento que proveen. Uno de esos aspectos clasifica a los mecanismos como implícitos o explícitos y el otro los clasifica como delegación o embedding. La combinación de esas variantes da lugar a cuatro categorías de lenguajes basados en prototipos. La diferenciación entre implícito y explícito es básicamente la misma que la del Tratado de Orlando. La clasificación según embedding o delegación depende de si los atributos que se comparten se hacen parte del objeto que "toma prestado" o se mantienen en el que los define, de donde son accedidos vía indirección. También distinguen entre delegación dinámica y estática según si las ligaduras a los objetos parent quedan fijas o se pueden cambiar dinámicamente.

La clasificación de sharing que presentamos en esta sección puede verse como una combinación de las ideas planteadas en los dos trabajos descriptos.

Existen cuatro dimensiones fundamentales sobre las que se debe decidir al compartir comportamiento. Las decisiones que se adopten en cada una de esas dimensiones son ortogonales, es decir que cualquier combinación de ellas da lugar un esquema de sharing válido. Se puede compartir comportamiento vía delegación o embedding, por objetos o por grupos de objetos, en forma implícita o explícita y estática o dinámicamente. El siguiente cuadro muestra la clasificación:



A continuación explicaremos cada una de las opciones definidas. Llamaremos donor al objeto que define el comportamiento que se comparte y borrower al objeto que lo reusa.

Delegación - embedding

La primer dimensión se refiere a dónde se mantiene el comportamiento común. Decimos que el sharing es por embedding cuando el comportamiento compartido se incorpora como parte del objeto que lo reusa. Es decir que se genera una copia del comportamiento común en el borrower. Así, cada vez que un objeto reciba un mensaje cuya respuesta comparte por embedding, puede responder en forma autónoma, sin necesidad de interactuar con el donor.

Supongamos que definimos un punto *p* en un espacio de dos dimensiones, con mensajes *x* e *y* que mantienen el valor de sus coordenadas. Luego queremos generar otro punto *q* reusando la definición de *p*, para ello hacemos que *q* comparta los mensajes con *p*. Pero este nuevo punto deberá mantener su posición en forma independiente, de manera que si *p* modifica sus coordenadas esos cambios no lo afecten. Entonces, el tipo de sharing que se debe usar es por embedding.

Un ejemplo muy común de sharing por embedding son las variables de instancia que proveen algunos lenguajes de programación. Las variables de instancia, si bien son comunes para todas las instancias de una clase, tienen asociado un valor que es local a cada objeto, forman parte de su estado independiente.

El sharing es por delegación cuando el comportamiento común es mantenido por el objeto donor. De esta manera, cada vez que el objeto que reusa recibe un mensaje que delega deberá interactuar con el donor para que éste responda como si fuera el receptor original. Una consecuencia inmediata de que el comportamiento común se mantenga en el donor es que si éste modifica ese comportamiento, los cambios se reflejan también en objeto borrower. Esto determina una diferencia importante con respecto a embedding, donde la interacción entre los objetos se hace solamente al momento de establecer la relación de sharing, y las modificaciones posteriores no son compartidas.

En el ejemplo anterior supongamos que *p* tiene un mensaje *up*, en respuesta al cual se desplaza incrementando el valor de su coordenada *y*. Para que *q* exhiba este comportamiento, puede definir el mensaje *up* y delegar la respuesta en *p*. De este modo, cada vez que *q* reciba ese mensaje, interactuará con *p* para que se ejecute el comportamiento correspondiente como si fuera propio de *q*, es decir se incremente el valor de la coordenada *y* de *q*. Supongamos que luego de esto *p* modifica su respuesta asociada al mensaje *up* para que ahora incremente en dos el valor de la coordenada *y*. En adelante, como *up* se comparte por delegación, cada vez que *q* reciba el mensaje también se sumará dos a su posición en el eje *y*.

En algunos de los lenguajes basados en clases se puede observar un esquema de sharing que combina embedding y delegación. Por ejemplo en Smalltalk [Gold89] las clases definen variables de instancia que son compartidas por embedding, y métodos que son compartidos por delegación. Cuando una instancia recibe un mensaje, lo delega en su clase para que se ejecute el método asociado en el contexto local de la instancia.

Por grupo - por objeto

Esta dimensión hace referencia al número de objetos que compartirán el comportamiento. Un esquema de sharing es por grupo cuando el comportamiento definido es compartido por un grupo de objetos.

El ejemplo más claro de sharing por grupo existente en los lenguajes de programación son las clases. Al definir una clase se especifica el comportamiento que compartirá un grupo de objetos: las instancias de la clase.

La alternativa es definir comportamiento individualmente para cada objeto. Los lenguajes basados en prototipos proveen un ejemplo claro de sharing por objeto. En un esquema de prototipos se debe definir el comportamiento para cada objeto creado, en general no existe manera de especificarlo de una sola vez para un conjunto de objetos.

En la modelización de los puntos *p* y *q* se usó sharing por objeto, porque el comportamiento y las relaciones de sharing se especificaron individualmente para cada objeto. Una forma alternativa sería definir una clase punto que contenga los mensajes correspondientes, y crear a partir de ella dos instancias *p* y *q*. En ese caso el comportamiento se define una vez para todos los puntos, es decir, el sharing es por grupo.

Explícito - implícito

La decisión sobre si un esquema de sharing es explícito o implícito alude a la forma de especificar los mensajes a compartir. Cuando el objeto que reusa puede especificar *por cada mensaje* un objeto donor con el que compartirá la respuesta a ese mensaje, decimos que el sharing es explícito. Así un objeto podría tener tantos donors como mensajes en su protocolo (sólo tendría que designar un objeto donor diferente para cada uno de los mensajes que conoce).

En el ejemplo, el punto *q* reusa del punto *p* la forma de responder al mensaje *up*. Si definimos otro punto de tres dimensiones *3dp* que tiene mensajes *x*, *y*, *z*, podríamos hacer que *q* se mueva también en un espacio tridimensional, definiéndole un mensaje *z*, cuya respuesta se comparte con el nuevo punto *3dp*. El tipo de sharing es explícito porque *q* está especificando "por mensaje", con quién comparte el comportamiento.

Decimos que el sharing es implícito cuando el objeto borrower tiene un donor "por defecto" de quien reusa el comportamiento. Es decir que cada vez que el objeto recibe un mensaje para el que no tiene definida una respuesta propia, "toma prestado" el comportamiento de su donor implícito. Ese donor implícito es usualmente llamado parent. Algunos lenguajes permiten especificar más de un parent (resultando, por ejemplo, en herencia múltiple).

Un tipo de sharing implícito podría usarse en el ejemplo para modelizar a partir de *p*, un punto de dos dimensiones con color. Definimos un objeto *cp* que tiene como donor implícito al punto *p*, y además define un mensaje propio *color*. Si *cp* recibe el mensaje *color* responderá con su comportamiento propio, pero si recibe cualquier otro mensaje se comportará como *p*, como un punto de dos dimensiones.

Distintas formas de sharing implícito pueden encontrarse en lenguajes basados en prototipos. El cloning por ejemplo, es un mecanismo que permite creación de objetos vía embedding implícito, porque crea un objeto a partir de otro, copiándolo en forma completa, sin necesidad de especificar individualmente cada uno de los mensajes. En el lenguaje SELF, todos los objetos tienen un link a un objeto parent en quien delegan implícitamente.

En los lenguajes basados en clases también se utiliza sharing implícito: la clase de un objeto actúa como su donor implícito, ya que en ella delega todo método que recibe (y comparte por

embedding todas las variables de instancia). En una jerarquía de herencia, la superclase de una clase también es su donor implícito, porque en el "method lookup" todo mensaje recibido que no implemente la clase es delegado a su superclase.

Un tipo de sharing explícito es el que proveen los sistemas que implementan el modelo actor [SLU88]. Presentan delegación explícita a través de protocolos especiales de pasaje de mensajes que requieren designar explícitamente al receptor como parte del mensaje. Una implementación concreta de estos sistemas es el lenguaje Delegation [SLU88], que provee mecanismos de sharing explícitos e implícitos.

Estático - dinámico

Esta última dimensión se refiere a la capacidad que tengan los objetos para modificar las relaciones de sharing establecidas con otros objetos. El sharing es dinámico si los objetos tienen la posibilidad de cambiar sus relaciones de sharing. En un esquema de sharing dinámico un objeto puede por ejemplo cambiar de donors, dejar de compartir con un donor para implementar ese comportamiento por sí mismo, o comenzar a tomar prestado de otro objeto un comportamiento que antes definía por sí mismo.

En SELF por ejemplo, las relaciones de sharing son dinámicas. Los atributos de los objetos, incluso la referencia al parent, se representan usando slots cuyo contenido puede modificarse en cualquier momento. Hybrid [Mercado88] es un lenguaje que implementa las relaciones típicas de los esquemas basados en clases pero con mayor flexibilidad. Como una forma de flexibilidad tiene por ejemplo, la posibilidad de modificar dinámicamente las jerarquías de herencia.

La alternativa es el sharing de tipo estático. El sharing es estático cuando las relaciones de sharing establecidas entre los objetos no pueden cambiar. La forma de compartir el comportamiento se fija en el momento de creación del objeto y se mantienen iguales mientras el objeto exista.

El esquema de sharing de Smalltalk, por ejemplo, es fundamentalmente estático porque las jerarquías de herencia no se pueden modificar, y si bien existen mecanismos para cambiar la clase de un objeto, su uso no es aconsejable.

Para poder representar en un marco uniforme todas las características del paradigma de Orientación a Objetos es importante disponer de una clasificación de sharing, porque es una de las características más relevantes del paradigma. La caracterización que presentamos en esta sección permite distinguir cuáles son los aspectos básicos que es indispensable proveer para poder implementar todos los mecanismos de sharing existentes.

Capítulo 4

Formalizaciones de la Orientación a Objetos

4. Formalizaciones de la Orientación a Objetos

Existen diversos trabajos sobre formalización de los conceptos del paradigma de Orientación a Objetos. Los primeros trabajos en esta área se concentraron en definir la semántica de la herencia entre clases, considerando a las mismas como construcciones básicas dentro del modelo. Se utilizan álgebras, teorías lógicas y otras notaciones formales (por ejemplo, cálculo lambda) para dar semántica a modelos de objetos. Más recientemente, se desarrollaron teorías cuyas construcciones primitivas son los objetos y no las clases. Estos trabajos proponen diversos cálculos basados en objetos para modelizar los lenguajes orientados a objetos, así como el cálculo lambda modela los lenguajes funcionales. La mayor parte del esfuerzo en el campo de fundamentos de los lenguajes orientados a objetos se refiere al desarrollo de sistemas de tipos para estos lenguajes.

En este capítulo comentaremos trabajos representativos de los diferentes acercamientos a una formalización del paradigma y analizaremos su adecuación al enfoque minimal descripto en el capítulo anterior.

Uno de los trabajos que utilizan álgebras para dar semántica a un modelo de objetos basado en clases es [BW]. En él se modelan las nociones de tipo, subtipo y herencia en base a la noción de order sorted algebra generalizada. El comportamiento se define por grupos y no por objetos. Utilizan una noción de tipo muy fuerte, que involucra al comportamiento. Se analiza y categoriza la idea de subtipo; un tipo es subtipo de otro si tiene comportamiento compatible. Se definen y representan subtipos parciales y subtipos totales usando order sorted algebras generalizadas. En este trabajo se resuelve de manera precisa la representación de herencia de forma de asegurar que el comportamiento de un objeto de una clase se comporta en forma similar a un objeto de la superclase.

Otro trabajo con este tipo de acercamiento es [BZ89], donde se presenta una semántica algebraica composicional para modelar características del paradigma de Orientación a Objetos. Se definen clases de objetos usando el lenguaje de especificación algebraica ASL [Wir86]. Cada clase de objetos es interpretada como la clase de las álgebras descriptas por la especificación algebraica correspondiente. Nuevas clases pueden crearse a partir de otras existentes usando combinadores de clases (entre los que se distinguen herencia y usa-a) que son interpretados como combinadores de especificaciones algebraicas. La clase especifica los atributos, métodos y threads de los objetos. Distinguen entre clases activas y clases pasivas, según tengan o no definido un thread. Representan la concurrencia basándose en sistemas de transición algebraicos.

Estas son dos ejemplos de una línea de trabajos cuya idea es representar a los objetos con técnicas similares a las usadas para especificar tipos de datos abstractos. Estos trabajos describen en forma denotacional el comportamiento de grupos de objetos, pero no representan explícitamente a objetos y mensajes, descuidando aspectos dinámicos de un sistema.

Entre los trabajos que usan lógica para dar semántica a un modelo de objetos basado en clases podemos mencionar al lenguaje Maude [MW91]. Maude es un lenguaje paralelo basado en lógica de reescritura. Los axiomas de esta lógica son reglas de reescritura. La reescritura concurrente coincide con la deducción en la lógica de reescritura. El objetivo del lenguaje Maude es tratar la programación concurrente y la programación orientada a objetos como programación declarativa, e integrarlas con la programación funcional. El lenguaje incluye módulos funcionales, de sistema y orientados a objetos. Las operaciones en los módulos se representan como reglas de reescritura. Un modulo orientado a objetos describe una clase, y los

métodos son reglas que permiten reescribir términos objeto. Un termino objeto contiene el identificador del objeto, la clase a la que pertenece, los nombres de los atributos del objeto y sus correspondientes valores. Maude soporta herencia entre clases interpretando la declaración de una subclase en un módulo orientado a objetos como la declaración de un subsort en el módulo de sistema que corresponde a la traducción de la subclase. Las reglas de reescritura de la clase se generalizan de modo de poder ser aplicadas a términos objetos pertenecientes a sus subclases.

En [AR90] se usa una notación similar a CSP para dar semántica a POOL, un lenguaje orientado a objetos paralelo. En POOL cada objeto es un proceso, cuyas acciones observables son el envío y la recepción de mensajes. Los objetos se ejecutan en forma concurrente y se sincronizan al interactuar por medio de los mensajes. Cuando un objeto envía un mensaje a otro, permanece inactivo hasta recibir la repuesta a dicho mensaje. Los objetos tienen un estado interno y métodos que se ejecutan al recibir los mensajes. El estado interno sólo puede ser manipulado por el mismo objeto, y las acciones sobre el estado no son observables desde el exterior del objeto. La recepción de los mensajes se realiza en forma explícita. Los objetos se crean a partir de clases, que describen el conjunto de mensajes y métodos y el cuerpo o actividad local del objeto que comienza cuando el objeto es creado. El cuerpo del objeto y los métodos se definen mediante sentencias y expresiones.

Se han desarrollado para POOL semánticas formales operacionales y denotacionales.

Loom [BPF97] y PolyToil [BSG95] son los últimos resultados de una serie de lenguajes orientados a objetos. El objetivo es proveer el poder expresivo que dan los sistemas de tipos, que tienen mecanismos complejos como el subtipado, pero con lenguajes más simples. Loom y PolyToil se definen a partir de cálculos, con reglas para chequeo de tipos y con semántica operacional (con reglas para reducción de términos). Son lenguajes basados en clases, estáticamente tipados. Ambos definen a las clases y tipos como construcciones básicas, por lo que pueden ser pasados como parámetros y ser devueltos como resultado de funciones. Definen la relación de matching entre tipos, que pretende ajustarse mejor a la relación de jerarquía de clases que la relación de subtipado, que en ciertos casos es demasiado restringida. Luego introducen un tipo de polimorfismo ligado a relación de matching, que provee mayor flexibilidad que el polimorfismo por subtipado.

Ambos lenguajes tienen definida una semántica imperativa que utiliza un store global para resultados parciales y finales de computo, y entornos locales para la ligadura de variables. El store puede almacenar funciones, registros, clases, valores de objetos, tipos básicos y direcciones del store.

Estos lenguajes definen una amplia gama de construcciones básicas. Presentan categorías sintácticas para bloques (de sentencias), expresiones (que incluyen funciones, registros, definiciones de clases, de herencia, constructor new y otros) y sentencias imperativas (que incluyen asignación, secuencia, condicional y loop). Esta variedad de términos básicos determina una serie extensa de reglas de semántica, algunas de las cuales son muy complejas.

[CP89] utiliza el cálculo lambda para dar una semántica denotacional de la herencia. Define la herencia como un mecanismo para derivar versiones modificadas de estructuras recursivas. Los objetos se modelan con records cuyos campos representan métodos. Una clase es una función generadora que crea objetos. Estas funciones generadoras se expresan usando cálculo lamba. Como los objetos se autoreferencian (via el uso de self) son representados como el punto fijo del generador. La herencia se modela como una operación entre generadores. Encuentra formalización de la herencia en la que están considerados tanto las referencias a self como a super. La correctitud de este modelo se prueba por comparación con la semántica usual del method lookup.

Los trabajos descriptos hasta aquí definen como básica la noción de clase. Algunos de ellos definen además primitivas de control. Por lo tanto no son adecuados para el modelo minimal que estamos buscando. A continuación describiremos algunos trabajos que proponen cálculos cuya construcción básica son los objetos.

Así como Cook encontró una semántica denotacional para la herencia de clases, [MVM96] describieron en forma denotacional la semántica de la delegación, usando como base el cálculo lambda. Desarrollaron un cálculo de objetos con el objetivo de solucionar algunos conflictos existentes entre la extensión de objetos y el encapsulamiento en los lenguajes basados en prototipos. Para eso, el cálculo distingue sintácticamente entre objetos y entidades heredables, a las que llama generadores. Los objetos son las entidades de primera clase: pueden recibir mensajes y ser pasados como parámetros. Un generador es un molde para un objeto. El generador más simple es la descripción de un método. Los generadores sólo pueden componerse. El cálculo define un operador para crear objetos a partir de un par de generadores, que puede interpretarse como un operador de delegación.

Entre los trabajos que utilizan cálculos para modelar el paradigma, uno de los más difundidos y que sirvió de referente para muchos otros estudios posteriores es el lambda cálculo de objetos de [MHF94]. Este trabajo extiende el lambda calculo no tipado con primitivas para la definición de objetos, envío de mensajes, modificación y extensión de objetos. El objetivo del trabajo es representar la especialización de métodos en la herencia. Modela lenguajes basados en prototipos.

La delegación entre objetos se representa a través de la modificación/extensión funcional de objetos existentes con nuevos mensajes. Los métodos de un objeto se representan a través de λ-abstracciones, donde el primer parámetro es el objeto receptor.

El cálculo tiene una semántica operacional basada en reglas de reducción, que definen la modificación y extensión de objetos, realizan β-reducción, y transformación de objetos a una forma standard.

Provee un sistema de tipos básico para el calculo. El tipo de un objeto define su interface. El sistema de tipos permite la especialización del tipo de los métodos al ser heredados. Se prueba la consistencia del sistema de tipos con respecto a las reglas de evaluación.

Martín Abadi y Luca Cardelli [AC96] definen una serie de cálculos para representar la esencia del paradigma de Orientación a Objetos.

Algunos de sus cálculos tienen definida semántica funcional y otros imperativa. Para ambos casos se desarrollan sistemas de tipos.

En estos cálculos, cada término representa un objeto, no hay otro tipo de entidades. Un objeto puede ser modificado, puede recibir un mensaje, o ser copiado, siempre dando como resultado otro objeto. El pasaje de parámetros no es primitivo, ni las funciones lo son. Tampoco hay mecanismos primitivos de sharing. No hay encapsulamiento, un objeto puede acceder a cualquier atributo de otro objeto.

Si bien hay aspectos importantes que no están resueltos, estos cálculos capturan las ideas básicas del paradigma y dado que sólo representan objetos y mensajes como entidades primitivas, se adecúan mas que cualquier otro formalismo al enfoque minimal que estamos buscando. Por estas razones elegimos uno de estos cálculos como punto de partida para nuestro trabajo.

Capítulo 5

El impς-cálculo de Abadi y Cardelli

5. El impς-cálculo de Abadi y Cardelli

Entre los acercamientos estudiados, el cálculo imperativo impς-cálculo, desarrollado por Martín Abadi y Luca Cardelli, es el que mejor se adecúa al enfoque minimalista que estamos buscando; por esta razón basamos en él nuestro trabajo.

En la primera sección de este capítulo describiremos el impς-cálculo. Presentamos su sintaxis y su semántica y analizamos si en este cálculo están representados los elementos y mecanismos básicos descriptos en el capítulo anterior.

En la segunda sección veremos cómo se pueden representar clases y herencia en base a este cálculo.

5.1 El impς-cálculo

En su libro 'A Theory of objects' Abadi y Cardelli presentan una familia de cálculos para representar el paradigma de orientación a Objetos. Los elementos básicos representados en estos cálculos son objetos y mensajes; las clases no son construcciones primitivas. Entre ellos, el ς-cálculo y el impς-cálculo son cálculos sencillos y sin tipos que sirven de base a los cálculos tipados más complejos.

La principal diferencia entre el ς-cálculo y el impς-cálculo es que la semántica del primero es funcional y la del segundo es imperativa. El primero es usado principalmente como base para cálculos tipados, tanto en el libro de Abadi y Cardelli como en la literatura posterior [AC96] [Liq97]. El segundo permite representar en forma natural efectos laterales tal como se encuentran en los lenguajes orientados a objetos, y mantener la identidad de los objetos. Por estas dos razones elegimos el impς-cálculo, que describiremos en detalle a continuación.

La sintaxis del impς-cálculo

Un objeto esta representado en el cálculo por la colección de mensajes que entiende. Asociado a cada mensaje hay un método. Un objeto que entiende los mensajes m_1, m_2,.....m_n y cuyos métodos asociados son b_1, b_2,.....b_n es representado por el término

$$[m_1 = \varsigma(x_1)b_1, m_2 = \varsigma(x_2)b_2, \ldots\ldots, m_n = \varsigma(x_n)b_n],$$

donde $\varsigma(x_i)b_i$ significa que en el cuerpo del b_i la variable x_i representa al receptor del mensaje m_i; b_1, b_2,.....b_n también son objetos.

La noción de variable de instancia no esta representada en el cálculo en forma directa, pero pueden ser representadas como casos particulares de mensajes. Cada variable se representa con un mensaje, cuyo método simplemente almacena un objeto.

El envío de un mensaje m_i a un objeto o se representa por

$$o.m_i.$$

Los mensajes inicialmente no tienen parámetros, pero estos pueden ser implementado.

El término

$$o.\, m \Leftarrow \varsigma(x)b$$

representa la modificación de la respuesta al mensaje m en el objeto o. El nuevo método asociado a m en o es b.

La operación

clone(o)

produce un nuevo objeto con los mismos mensajes y métodos que el objeto o. Esta operación puede ser implementada en base a las otras.

El término

let x = o in b

representa al objeto b en el que las apariciones de la variable x se ligan con el resultado de evaluar o.

La sintaxis completa del cálculo aparece en la Figura 1.

No se diferencia entre dos objetos que tengan definidos los mismos mensajes y métodos pero en distinto orden.

Sintaxis completa del Impς-cálculo

o,b ::=

$[m_i = \varsigma(x_i)b_i, i \in 1..n]$	Objeto
x	Variable
$o.l$	Envío de mensaje l al objeto a
$o.l \Leftarrow \varsigma(x)b$	Modificación del método asociado al mensaje
$clone(o)$	Copia del objeto o
$let\ x = o\ in\ b$	Definición local

Figura 1

Por ejemplo, suponiendo que tenemos una representación para los enteros, escribamos en el cálculo un objeto que represente el punto (5,6) en el plano:

$[x = \varsigma(self_1)5,$

$y = \varsigma(self_2)6]$

Un punto-móvil, esto es, un punto que sepa desplazarse en el plano, se podría representar de esta manera:

$[x = \varsigma(self_1)5,$

$y = \varsigma(self_2)6,$

$arriba = \varsigma(self_3)\ self_3.y \Leftarrow self_3.y + 1,$

$abajo = \varsigma(self_4)\ self_4.y \Leftarrow self_4.y - 1,$

$izquierda = \varsigma(self_5)\ self_5.x \Leftarrow self_5.x - 1,$

$derecha = \varsigma(self_6)\ self_6.x \Leftarrow self_6.x + 1,$

$invertir = \varsigma(self_7)\ let\ a = clone(self_7)\ in\ (self_7.x \Leftarrow a.y).y \Leftarrow a.x]$

Los mensajes up, down, right y left hacen que el punto se mueva en sus coordenadas x e y, cuando el punto recibe el mensaje invertir se intercambia los valores de sus coordenadas x e y.

Variantes

Abadi y Cardelli definen algunas variantes de éste cálculo. Una de ellas es el impqrcalculus, en el que se diferencian sintácticamente las variables de instancia (llamadas campos por Abadi y Cardelli) de los métodos. El término que representa el update toma la forma de una asignación cuando se refiere a variables de instancia, y no a métodos propios. Las variables de instancia de un objeto son evaluadas antes que el resto del objeto. Presentan una traducción del impq f-cálculo al impq-cálculo en la que la evaluación previa se realiza con la construcción let.

Otra variante es el imp^-cálculo, que es ^.-cálculo con efectos laterales y pasaje de parámetros por valor. Tiene abstracción, aplicación de funciones y asignación a variables ligadas a un *X.* También este cálculo es traducido al impqrcálculo. Esta traducción es muy útil porque permite introducir funciones con parámetros al calculo original. Usando estas funciones se define el pasaje de mensajes con argumentos.

Usando una combinación del impA,-cálculo y el impqr calculus, llamada imp^qr calculus, presentan una implementación de los booleanos y los números naturales.

Semántica Qperacional

La semántica operacional describe la reducción de términos en base a un almacenamiento global que llamaremos **store.**

Cada término reduce a un **valor** o **resultado,** v ::= *[l¡ = i¡ tel"n],* que es un conjunto de nombres de mensaje, cada uno de ellos con una **locación** o dirección del store asociada, donde se encuentra el método correspondiente.

Cada locación i, en el store contiene una **method closure,** que es un par *<g(x,)b¡, S>* donde el objeto b_i representa un método en el que la variable x_i denota al objeto al que pertenece el método y *S* es una **pila** o stack que mapea locaciones a method closures. Esta pila es usada en la evaluación del método, y contiene las ligaduras de variables (ver Figura 2).

Una expresión de la forma

$$a.\ S \vdash b \rightsquigarrow v.\ \&\ ,$$

donde a y a ' son stores, S es una pila, b es un término y v un valor, se denomina juicio de reducción de términos, y significa que en el store a y en la pila S el término b evalúa al resultado v, produciendo un nuevo store modificado a '.

La relación ~> se denomina reducción y esta definida inductivamente por las reglas de la Figura 3.

Los stores se representan como secuencias finitas que mapean locaciones a method closures. La secuencia i; <- m, representa un store que mapea cada locación i,- a la closure m,. La expresión a.i* <—m* significa que en la locación *ik* del store a se almacena la closure *mk.*

Las stacks también se representan como secuencias finitas, y mapean variables a valores.

Las reglas **(Store 0), (Store i), (Stack 0)** y **(Stack x)** describen la formación de stacks y stores. **Store 0** y **Stack 0** significan que el stack vacío es un stack bien formado y el store vacío es un store bien formado. Las reglas **(Store i)** y **(Stack x)** representan la incorporación de nuevos elementos al stack y al store.

Según la regla (Red x) una variable reduce al resultado que tiene asociado en el stack corriente.

La regla (**Red Object**) determina que un término que describe un objeto en esta forma reduce a una colección de nuevas locaciones en el store, y que el store es extendido con ellas. Cada una de esas nuevas locaciones contiene el método asociado a un mensaje.

La regla (**Red Select**) determina que cuando se envía un mensaje a un objeto, el objeto se reduce a un resultado y el método asociado al mensaje enviado es evaluado. En la evaluación del método la variable self se liga en la pila corriente con el valor que se obtuvo de evaluar el objeto.

Es importante notar que el store contiene términos que no están totalmente evaluados, las respuestas a los mensajes son objetos que se evalúan en el momento en que se recibe el mensaje.

La modificación de un método de un objeto se evalúa reduciendo el objeto a un resultado y modificando la locación en el store correspondiente a ese método con el nuevo method closure (**Red Update**).

Al clonar un objeto se reduce el objeto original a un resultado y se crea un nuevo objeto alocando nuevos lugares en el store. Estas nuevas locaciones se asocian a copias de las method closures del objeto original (**Red Clone**).

Según (**Red Let**) para evaluar un término let primero se evalúa el término asociado a la variable ligada obteniendo un resultado y un nuevo store, y luego se evalúa el cuerpo del let en una stack donde el resultado anteriormente obtenido esta ligado a la variable.

ι	Locación en el store
$V ::= [l_i = \iota_i{}^{i \in 1..n}]$	Resultado
$\sigma ::= \iota_i \rightarrow <\varsigma(x_i)b_i , S_i >^{i \in 1..n}$	Store
$S :: x_i \rightarrow v_i{}^{i \in 1..n}$	Stack
σ |- <>	Juicio de store bien formado
$\sigma . S$ |- <>	Juicio de stack bien formado
$\sigma . S$ |- a ~> $v . \sigma'$	Juicio de reducción de términos

Figura 2

(Store ∅)

$$\frac{}{\emptyset \vdash <>}$$

(Store ι)

$$\frac{\sigma \,.\, S \vdash <> \quad \iota \notin dom(\sigma)}{\sigma,\ \iota \rightarrow <\varsigma(x)b\,,\ S> \vdash <>}$$

(Stack ∅)

$$\frac{\sigma \vdash <>}{\sigma \,.\, \emptyset \vdash <>}$$

(Stack x) (l_i, ι_i distintos)

$$\frac{\sigma \,.\, S \vdash <> \quad \iota_i \in dom(\sigma) \quad x \notin dom(S) \quad \forall i \in 1..n}{\sigma \,.\, (S,\ x \rightarrow [l_i = \iota_i{}^{\,i \in 1..n}]) \vdash <>}$$

(Red x)

$$\frac{\sigma \,.\, (S',\ x \rightarrow v,\ S'') \vdash <>}{\sigma \,.\, (S',\ x \rightarrow v,\ S'') \vdash x \sim> v \,.\, \sigma}$$

(Red Object) (l_i , ι_i distintos)

$$\frac{\sigma \,.\, S \vdash <> \quad \iota_i \notin dom(\sigma) \quad \forall i \in 1..n}{\sigma \,.\, S \vdash [l_i = \varsigma(x_i)b_i] \sim> [l_i = \iota_i{}^{\,i \in 1..n}] \,.\, (\sigma, \iota_i \rightarrow <\varsigma(x_i)b_i,\ S>{}^{\,i \in 1..n})}$$

(Red Select)

$$\frac{\sigma \,.\, S \vdash a \sim> [l_i = \iota_i{}^{\,i \in 1..n}] \,.\, \sigma' \qquad \sigma'(\iota_j) = <\varsigma(x_j)\, b_j\ ,\ S'> \qquad x_j \notin dom(S') \quad j \in 1..n \quad \sigma \,.\, (S', x_j \rightarrow [\, l_i = \iota_i{}^{\,i \in 1..n}\,]) \vdash b_j \sim> v \,.\, \sigma''}{\sigma \,.\, S \vdash a.l_j \sim> v \,.\, \sigma''}$$

(Red Update)

$$\frac{\sigma \,.\, S \vdash a \sim> [l_i = \iota_i{}^{\,i \in 1..n}] \,.\, \sigma' \qquad j \in 1..n \quad \iota_j \in dom(\sigma')}{\sigma \,.\, S \vdash a.l_j \Leftarrow \varsigma(x)b \sim> [l_i = \iota_i{}^{\,i \in 1..n}] \,.\, (\sigma'.\ \iota_j \leftarrow <\varsigma(x)b\,,\ S>)}$$

(Red Clone) ι_i' distintos

$$\frac{\sigma \,.\, S \vdash a \sim> [l_i = \iota_i{}^{\,i \in 1..n}] \,.\, \sigma' \quad \iota_i \in dom(\sigma') \quad \iota_i' \notin dom(\sigma') \quad \forall\, i \in 1..n}{\sigma \,.\, S \vdash clone(a) \sim> [l_i = \iota_i'{}^{\,i \in 1..n}] \,.\, (\sigma', \iota_i' \rightarrow \sigma'(\iota_i)^{\,i \in 1..n})}$$

(Red Let)

$$\frac{\sigma \,.\, S \vdash a \sim> v' \,.\, \sigma' \qquad \sigma'.\ (S,\ x \rightarrow v') \vdash b \sim> v'' \,.\, \sigma''}{\sigma \,.\, S \vdash let\ x{=}a\ in\ b \sim> v'' \,.\, \sigma''}$$

Figura 3

Un ejemplo que no puede faltar: la pila

Veamos como representar una pila en el cálculo. Una *pila* entiende un mensaje *apilar* para agregar nuevos elementos a la *pila*, un mensaje *desapilar* para sacar el elemento que almacena en el tope y un mensaje *vacía* que devuelve *true* si la pila esta vacía y *false* en caso contrario. Asumiremos que *true* y *false* son constantes; su comportamiento puede ser implementado en el cálculo.

La *pila* interactúa con el primero de una secuencia de nodos de pila, que guardan los elementos.

pila ≡ *[nodos= ς(self) body-pila1,*
apilar= ς(self) body-pila2,
desapilar= ς(self) body-pila3,
vacía= ς(self) body-pila4]

Cada *nodo-pila* no vacío interactúa a otro, que tiene que tiene el siguiente elemento. Uno *nodo-pila* puede responder al mensaje *vacío* que indica si tiene un elemento almacenado o no.

nodo-pila ≡ *[vacío = ς(self) body-b1,*
elemento= ς(self) body-b2,
siguiente= ς(self) body-b3]

Una *pila* inicialmente esta vacía, simbolizamos esto haciendo que conozca un *nodo-pila* que no tiene ningún elemento. Representamos un *nodo-pila* que no tiene ningún elemento haciendo que su *elemento* y su *siguiente* sean objetos que no pueden responder a ningún mensaje: *[]*.

pila ≡ *[nodos= ς(self) [vacío = ς(self) true,*
elemento= ς(self) [],
siguiente= ς(self) []],
apilar= ς(self) body-pila2,
desapilar= ς(self) body-pila3,
vacía= ς(self) body-pila4]

Cuando la pila vacía recibe el mensaje *apilar*, debe:

- almacenar en un nuevo nodo el nuevo elemento,
- hacer que su nodo siguiente sea el nodo que encabezaba la secuencia, y
- tomar al nodo recientemente creado como primer nodo de la secuencia.

Cuando recibe el mensaje *desapilar*, debe:

- tomar el elemento del *nodo-pila* que conoce, y
- eliminar un nodo de la secuencia, para esto toma como tope de pila al *nodo-pila* siguiente.

Cuando recibe el mensaje *vacía*, debe mandarle un mensaje al primer nodo de la secuencia para ver si tiene elementos o no.

```
pila ≡  [nodos= ς(self) [vacío = true,
                         elemento= ς(self) [],
                         siguiente= ς(self) []],
        apilar= ς(self) λ(nuevo-elemento)
                         let anterior = self.nodos in
                                 let ( x = [vacía = false,
                                            elemento= ς(self) nuevo-elemento,
                                            siguiente= ς(self) anterior] ) in
                                                 self.nodos ⇐ ς(self) x )
        desapilar= ς(self) let elem = nodos.elemento
                         in (let x = (self.nodos ⇐ ς(self) self.nodo.siguiente ) in elem)
        vacía = nodos.vacío
        ]
```

Adecuación del cálculo al modelo minimalista pleanteado

El cálculo de Abadi y Cardelli, a diferencia de la mayoría de los otros formalismos encontrados en la literatura, representa solo los elementos básicos del paradigma: objetos y mensajes.

Soporta también los mecanismos básicas de creación (mediante los términos [ ] y clone) y envío de mensaje.

La modificación de objetos es parcialmente provista. Permite cambiar el comportamiento de un objeto modificando la respuesta del objeto a un mensaje. Sin embargo, no permite extender el protocolo de un objeto, y este mecanismo es imposible de implementar con las otras operaciones provistas. Aunque los mensajes no pueden llevar parámetros, estos puede ser implementado representando los métodos como funciones, como se verá en la siguiente sección.

Los objetos en el cálculo tienen identidad, representada por un conjunto locaciones en el store. Un mismo objeto puede estar referenciado por otros objetos, y una modificación hecha sobre el objeto se percibirá desde sus distintas referencias; esta propiedad se conoce con el nombre de **integridad referencial**. Esta es una ventaja del cálculo elegido con respecto al cálculo funcional ς-calculus.

El polimorfismo es inherente al impς-calculus, ya que la noción de tipos no existe y no hay restricciones al respecto de que el mismo mensaje este en el protocolo de más de un objeto. Al enviar un mensaje a un objeto se responde con el comportamiento que esta definido en ese objeto en particular. Es importante notar que el late binding, como se presenta en los lenguajes orientados a objetos, también es una característica del cálculo (ver (**Red Select**)).

Encapsulamiento es una característica básica que el cálculo no posee. No hay forma primitiva de definir mensajes privados ni variables de instancia que sólo sean referenciadas por el objeto que las define; soluciones a este problema se encontraron al incorporar sistemas de tipos [Abadi96]. Además, cualquier objeto puede modificar el comportamiento de otro cambiando sus respuestas a mensajes mediante update, si conoce su protocolo. Una solución simplista sería añadir una restricción sintáctica que prohiba usar updates si no se refieren a la variable self, pero esto reduce considerablemente el poder expresivo del cálculo, ya que no se podrían expresar funciones.

Los objetos en el cálculo no tienen ningún mecanismo primitivo que le permita compartir comportamiento, este es el problema al que dedicamos gran parte de este trabajo.

Finalmente, una propiedad muy importante: Abadi y Cardelli mostraron como implementar el cálculo lambda usando este cálculo, lo que demuestra que es computacionalmente completo.

5.2 Representacion de mecanismos de sharing por grupo

Dado que los mecanismos para compartir comportamiento no son primitivos en el cálculo de Abadi y Cardelli, estos deben ser implementados en base a los conceptos mas simples. En esta sección veremos la implementación de clases propuesta por Abadi y Cardelli. Para ello estudiaremos la representacion de funciones y de premétodos. Un premétodo es una función que se utiliza para simular la evaluación de un mensaje sobre un receptor que no tiene definido el comportamiento efectivo para ese mensaje. Los premétodos son la base de las representaciones de los mecanismos de sharing.

Representación de funciones

Una función se representa en el cálculo con un objeto que tiene dos mensajes: arg y val. El mensaje arg representa el argumento de la función, y val el cuerpo. Dentro del cuerpo de la función el argumento se referencia enviándole a self el mensaje correspondiente al argumento.

Por ejemplo, una función que reciba un número x y devuelva $x + 2$, suponiendo que cuando el número x recibe el mensaje *succ* devuelve $x + 1$, se escribiría

$$[arg = \varsigma(x)[],\ val = \varsigma(x)((x.arg).succ).succ].$$

Inicialmente el valor del argumento es $[]$, un objeto que no entiende ningún mensaje.

Las funciones se incluyen en la sintaxis del calculo. $\lambda x.b$ representa la definición de una función con argumento x y cuerpo b. Si f es una función, la aplicación de f a un parámetro real a se escribe $f(a)$.

La abstracción $\lambda x.b$ se traduce:

$$<<\lambda x.b>> = [arg = \varsigma(x)[],\ val = \varsigma(x)<<b>>\ \{x \leftarrow x.arg\}],$$

donde $b\{x \leftarrow a\}$ significa que las ocurrencias de x en b se reemplazan sintácticamente por a.

La aplicación de la función a un parámetro real se realiza modificando la respuesta del mensaje *arg* y enviando al objeto función el mensaje *val* para forzar a su evaluación.

La aplicación $f(a)$ se traduce así:

$$<<f(a)>> = let\ x = <<a>>\ in\ ((clone(<<f>>).arg \Leftarrow \varsigma(y)\ x).val.$$

La presencia del *let* produce el efecto de pasaje de parámetro por valor.

El cuerpo de función b es clonado en cada aplicación para evitar interferencias con posteriores aplicaciones de la misma función.

Representación de premétodos

La importancia de los premétodos radica en que son la base para la implementación de los mecanismos de sharing.

Uno de los problemas que se encuentran al definir comportamiento compartido es referido a la ligadura de la variable self. Una aproximación ingenua al problema de compartir comportamiento es la siguiente: cuando un objeto borrower recibe un mensaje para el que no define comportamiento propio, lo reenvía a su donor. El donor responderá el mensaje de acuerdo al comportamiento que tiene definido. Pero esta solución tiene un problema: cada vez que ocurra self en el método definido por el donor naturalmente se referirá al donor, es decir que

al responder al mensaje éste usará sus propias variables de instancia. Este no es el efecto que queremos lograr al compartir comportamiento.

Los premétodos son métodos que abstraen la variable self, es decir, son funciones cuyo cuerpo representa el método en el donor, y cuyo argumento es el pretendido receptor original del mensaje. El cuerpo del premétodo referencia al argumento cada vez que el método original hubiera referenciado a la variable self.

Un forma de implementación del sharing será que los donors tengan definidos premétodos que contengan el comportamiento a compartir. Cuando un objeto quiere usar el comportamiento definido en un premétodo en un donor, lo invoca mediante un mensaje al donor enviándose a sí mismo como argumento.

Analicemos un ejemplo.

Supongamos que tenemos un objeto que representa una cuenta bancaria

[saldo= ς(self) s,
depositar= ς(self) λ(cantidad)
let x= self. saldo. sumar (cantidad) in (self. saldo ⇐ ς(self) x),
extraer= ς(self) λ(cantidad)
let x= self. saldo. restar (cantidad) in (self. saldo ⇐ ς(self) x)]

y queremos reusar el comportamiento definido para depositar y extraer. Vamos a crear un nuevo objeto que tiene definido este comportamiento en forma de premétodos, para que este comportamiento pueda ser prestado a los objetos que representen cuentas bancarias.

Llamaremos *trait* a una colección de premétodos. Llamaremos *trait-cuenta* a este objeto que abstrae el comportamiento común a las cuentas bancarias.

trait-cuenta ≡
[depositar= ς(self)λ(receptor)
λ(cantidad)
let x= receptor. saldo. sumar (cantidad)
in (receptor. saldo ⇐ ς(self) x),
extraer= ς(self)λ(receptor)
λ(canti3dad)
let x= receptor.saldo. restar (cantidad)
in (receptor. saldo ⇐ ς(self) x)]

Cada cuenta bancaria en particular estará representada por

[saldo= ς(self) s ,
depositar= ς(self) trait-cuenta.depositar (self),
extraer= ς(self) trait-cuenta.extraer (self)]

Cuando un objeto cuenta bancaria recibe el mensaje *depositar*, le envía a *trait-cuenta* el mismo mensaje enviándose a sí mismo como argumento. De esta manera se logra que al evaluar la respuesta al mensaje *depositar* definida en trait-cuenta se modifique el saldo de la cuenta que originalmente recibió el mensaje.

Representación de clases

Cada clase se representa con un objeto que tiene un método *new* que permite crear instancias de la clase y un trait que guarda el comportamiento común.

El método *new* creará una nueva instancia de la clase que use el comportamiento definido en el trait.

Por ejemplo, una clase CuentaBancaria se definiría así:

$$
\begin{aligned}
&\mathit{CuentaBancaria} \equiv \\
&\quad [\mathit{new}= \varsigma(\mathit{clase}) \\
&\qquad [\mathit{saldo}= \varsigma(\mathit{self})\ \mathit{clase.saldo}(\mathit{receptor}), \\
&\qquad \mathit{depositar}= \varsigma(\mathit{receptor})\mathit{clase.depositar}\ (\mathit{receptor}), \\
&\qquad \mathit{extraer} = \varsigma(\mathit{receptor})\mathit{clase.extraer}\ (\mathit{receptor}) \\
&\qquad] \\
&\quad \mathit{saldo}= \varsigma(\mathit{self})\ \lambda(\mathit{receptor})0, \\
&\quad \mathit{depositar}= \varsigma(\mathit{self})\lambda(\mathit{receptor}) \\
&\qquad\qquad\qquad \lambda(\mathit{cantidad}) \\
&\qquad\qquad\qquad\qquad \mathit{let}\ x= \mathit{receptor.\ saldo.\ sumar}\ (\mathit{cantidad})\ \mathit{in} \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{receptor.\ saldo} \Leftarrow \varsigma(\mathit{self})\ x\ ), \\
&\quad \mathit{extraer}= \varsigma(\mathit{self})\lambda(\mathit{receptor}) \\
&\qquad\qquad\qquad \lambda(\mathit{cantidad}) \\
&\qquad\qquad\qquad\qquad \mathit{let}\ x= \mathit{receptor.\ saldo.\ restar}\ (\mathit{cantidad})\ \mathit{in} \\
&\qquad\qquad\qquad\qquad\qquad (\ \mathit{receptor.\ saldo} \Leftarrow \varsigma(\mathit{self})\ x\ ) \\
&\quad]
\end{aligned}
$$

Al ser invocado, *new* crea un objeto que entiende los mensajes *saldo*, *depositar* y *extraer*. Cuando el nuevo objeto recibe uno de estos mensajes, el premétodo correspondiente se busca en la clase y se aplica al receptor original del mensaje.

En un objeto creado por esta clase podrá modificar su saldo (al recibir el mensaje *depositar*) y a partir de ese en ese momento la respuesta para el mensaje *saldo* ya no se tomará de la clase.

Los premétodos *depositar* y *extraer* son buscados en la clase cada vez que el objeto recibe esos mensajes. Si en algún momento el comportamiento asociado a depositar cambia en la clase Cuenta Bancaria, ese cambio se verá reflejado en cada instancia la próxima vez que reciba el mensaje depositar.

En general, una clase se define así:

$$
\begin{aligned}
&\mathit{let}\ c = [\mathit{new}= \varsigma(\mathit{clase})\ [m_i = \varsigma(\mathit{rec})\ \mathit{clase}.m_i(\mathit{rec})^{\,i \in 1..n}] \\
&\qquad m_i = \varsigma(x_i)\ \lambda(\mathit{receptor})b_i^{\,i \in 1..n}]
\end{aligned}
$$

Una subclase *c'* de la clase *c* se puede representar con un objeto que reúsa los premétodos de *c* y agrega comportamiento nuevo.

$$
\begin{aligned}
&\mathit{let}\ c' = [\mathit{new}= \varsigma(\mathit{clase})\ [m_i = \varsigma(\mathit{rec})\ \mathit{clase}.m_i(\mathit{rec})^{\,i \in 1..n+m}], \\
&\qquad m_j = \varsigma(x_j)\ c.m_j^{\,j \in 1..n} \\
&\qquad m_k = \varsigma(x_k)\ c'.m_k^{\,k \in n+1..n+m}]
\end{aligned}
$$

La clase *c'* agrega m nuevos mensajes y define sus premétodos, pero utiliza los premétodos definidos en *c* en los n mensajes que hereda.

Si alguno de los premétodos definidos en la clases o en la subclase es modificado, este cambio se verá reflejado en cada una de las instancias.

Capítulo 6

Un Cálculo Imperativo con Extensión de Objetos

6 . Un Cálculo Imperativo con Extensión de Objetos

El impς-cálculo de Abadi y Cardelli no permite extensión de objetos. Como veremos en el Capítulo 7, esto constituye un problema al momento de representar determinados mecanismos de sharing. En este capítulo presentamos el impEς-cálculo, que tiene una semántica imperativa y provee la extensión como una operación primitiva sobre los objetos.

6.1 Extensión de objetos en el impς-cálculo

Como ya se mencionó en el Capítulo anterior, el impς-cálculo no permite extender dinámicamente el protocolo de los objetos. No existe un término básico que represente la acción de agregar nuevos mensajes a un objeto, ni hay posibilidad de implementarla utilizando las construcciones provistas. El conjunto de locaciones de memoria correspondiente a los mensajes de un objeto, se mantiene fijo durante toda la vida del objeto, si bien el contenido de esas locaciones puede modificarse. Esta decisión de diseño provee simplicidad al formalismo, porque la introducción de una operación de extensión de objetos requiere considerar ciertas dificultades.
Un problema al definir extensión de objetos es la posibilidad de colisión de un mensaje a agregar con alguno de los mensajes existentes en el objeto. En cálculos con tipos esto requiere llevar un control sobre los mensajes del tipo real del objeto a extender.
Otro problema a tener en cuenta es que al extender un objeto se debe modificar su representación para agregar el nuevo mensaje, pero manteniendo la identidad del objeto.
Supongamos que disponemos de un operador de extensión $\Leftarrow^+$ para agregar nuevos mensajes a los objetos en el impς-cálculo. El término $o.m \Leftarrow^+ \varsigma(self)b$ aloca una nueva dirección en el store para m y la agrega a la representación de o. La semántica del cálculo contaría así con la regla

(Red WrongExtend)

$$\frac{\sigma \bullet S \vdash a \rightarrow [l_i = \iota_i^{\,i \in 1..n}] \bullet \sigma' \qquad \iota_{n+1} \notin dom(\sigma')}{\sigma \bullet S \vdash a.l_{n+1} \Leftarrow^+ \varsigma(x)b \rightarrow [l_i = \iota_i^{\,i \in 1..n+1}] \bullet (\sigma', \iota_{n+1} \rightarrow <\varsigma(x)b, S>)}$$

En este contexto consideremos el punto-movible presentado Capítulo 5, definido dentro del siguiente término

```
. . .
let punto = [ x = ς(self1 ) 5,
              y = ς(self2 ) 6,
              izquierda = ς(self3 ) let valx = (self3.x)  in  (self3 .x ⇐ ς(self1) valx - 1),
              derecha = ς(self4 ) let valx = (self4.x)  in  (self4 .x ⇐ ς(self1) valx + 1),
              arriba = ς(self5 ) let valy = (self5.y)  in  (self5 .y ⇐ ς(self2) valy + 1),
              abajo = ς(self6 ) let valy = (self6.y)  in  (self6 .y ⇐ ς(self2) valy - 1),
              invertir = ς(self7 ) let a = clone(self7) in (self7 .x ⇐ a.y ).y⇐ a.x  ]
   in let dibujo  = [unPixel = ς(self) punto, . . .]
   in let a  =  punto.color⇐+ ς(self) azul
   in  unBloque
. . .
```

En la evaluación de este término, supongamos que el objeto *punto* reduce al valor

$$v_{punto} \equiv [x = \iota_1,\ y = \iota_2,\ izquierda = \iota_3,\ derecha = \iota_4,\ arriba = \iota_5,\ abajo = \iota_6,\ invertir = \iota_7]$$

donde ι_1 , ... , ι_7 son locaciones de store. Al evaluar el objeto *dibujo*, se almacena en su representación (asociado al mensaje *unPixel*) una referencia a punto a través de v_{punto} . Luego se extiende el objeto *punto* para que tenga color. Usando la regla de evaluación para el operador de extensión definido anteriormente el punto extendido reduce ahora a

$$v_{puntoExt} \equiv [x = \iota_1,\ y = \iota_2,\ izquierda = \iota_3,\ derecha = \iota_4,\ arriba = \iota_5,\ abajo = \iota_6,\ invertir = \iota_7,\ color = \iota_8]$$

Supongamos que dentro de *unBloque* se necesita conocer el nuevo color del pixel del objeto *dibujo*. Para ello se debería enviar el mensaje

dibujo.unPixel.color

La evaluación de este envío de mensaje reduce primero el término *dibujo.unPixel*, obteniendo v_{punto}. A partir de este resultado intenta evaluar el método asociado a *color*. Pero se produce un error porque v_{punto} no contiene el mensaje color. Es decir, que si bien el objeto *punto* tiene un mensaje *color*, *dibujo* no puede enviárselo porque conoce al punto a través de un resultado evaluado antes de la extensión con el mensaje *color*. Observamos, sin embargo, que sí es posible obtener el color de punto a través de

a.color

ya que la referencia usada aquí para acceder a punto es $v_{puntoExt}$.
El problema surge porque al extender el objeto *punto* se alteró su representación y su identidad. En la semántica del impς-cálculo, la identidad de los objetos está representada también por el conjunto de locaciones de store correspondientes a sus mensajes. Por lo tanto agregar un mensaje a un objeto implica alterar su identidad, con lo que se pierde la integridad referencial, porque una extensión en el protocolo de un objeto no es percibida desde todos los otros objetos que lo referencian.
Es por eso que la extensión de objetos no se puede incorporar en forma directa al impς-cálculo.

6.2 El impEς-cálculo

En este trabajo definimos el impEς-cálculo, inspirado en el impς-cálculo, que permite la extensión de objetos con nuevos mensajes. La sintaxis del impEς-cálculo es similar a la del impς-cálculo. Tiene una semántica imperativa, en la que la extensión mantiene la identidad de los objetos.
En este cálculo definimos un operador para extender el protocolo de los objetos con nuevos mensajes. La extensión de objetos tiene dos variantes, que se diferencian por su efecto frente a la colisión de mensajes nuevos con mensajes existentes. Una de las variantes de extensión sobreescribe la versión existente del mensaje en colisión, y la otra conserva el mensaje existente ignorando la extensión.
Elegimos una representación para la identidad de los objetos que sea compatible con el operador de extensión adoptado. Es decir, definimos una semántica en donde la identidad de los objetos no se pierde al extender su protocolo con nuevos mensajes. A diferencia del impς-cálculo de

Abadi y Cardelli, en el imEς-cálculo se representa la identidad de los objetos en forma explícita y diferente de la representación propia del objeto. Para ello se define un store I de identificaciones de objetos, en donde existe una sola entrada por cada objeto. La identidad de un objeto es su entrada única en el store de identificaciones, que es siempre la misma durante toda la vida del objeto. El contenido de esa entrada es la representación propia del objeto, que puede variar con las modificaciones y extensiones que se hagan al objeto. Además, todo objeto es referenciado a través de su identidad, con lo que las operaciones de modificación y extensión no comprometen la integridad referencial.

Sintaxis

La sintaxis completa del impEς-cálculo es

o,b ::=

$[m_i = \varsigma(x_i)b_i^{\,i \in 1..n}]$	Objeto
$\underline{x}$	Variable
$o.m$	Envío de mensaje m al objeto o
$o.m \Leftarrow \varsigma(x)b$	Modificación del método asociado al mensaje m
$o.m \Leftarrow^{+} \varsigma(x)b$	Extensión del objeto o con mensaje m
$let\ x = o\ in\ b$	Definición local

Los términos para representar objetos, variables, envío de mensajes y definición local son iguales que en el impς-cálculo. La clonación de objetos no es un término primitivo en este cálculo. Pero eso no lo hace menos expresivo que el impς-cálculo, ya que en ambos se puede construir *clone* a partir de los otros términos básicos. La implementación del *clone* en el impEς-cálculo se verá en el Capítulo 7.
La extensión de objetos con nuevos mensajes se representa en el impEς-cálculo a través de los operadores $\Leftarrow+$ y $\Leftarrow$. El término

$$o.m \Leftarrow+ \varsigma(x)\ b$$

agrega al objeto o un nuevo mensaje llamado m, con respuesta asociada $\varsigma(x)\ b$, donde el objeto b representa al cuerpo del método y la variable x representa a self. Esta extensión es conservativa, en el sentido de que si el objeto o ya tuviera un mensaje llamado m la evaluación de la extensión no tiene ningún efecto.
El término

$$o.m \Leftarrow \varsigma(x)\ b$$

modifica en o el método asociado al mensaje m, si m pertenece al conjunto de mensajes de o (igual que la modificación de objetos en el impς-cálculo). Pero si m no pertenece al protocolo de o, este término agrega el mensaje m al objeto o con método asociado b y variable self x. Decimos que esta extensión es no conservativa, porque si el objeto ya tuviera el mensaje m se modifica la respuesta asociada.

Semántica operacional

Al igual que en el impς-cálculo, la semántica se define en términos de relaciones de reducción. Pero en el impEς-cálculo se utiliza un store global de objetos y un store global de identificaciones de objetos. La semántica operacional relaciona términos del cálculo con identificaciones de objetos. La relación:

$$(\sigma, I) \bullet S \vdash a \rightsquigarrow id \bullet (\sigma', I')$$

expresa que a partir de un par de stores (σ, I) de objetos y de identificaciones, y una pila S el término objeto a reduce a su identificación id produciendo un par de stores modificados σ' e I'. Las entidades utilizadas en la semántica son:

ι	locación en el store
$v ::= [l_i = \iota_i^{\,i \in 1..n}]$	resultado
$\sigma ::= \iota_i \rightarrow \langle \varsigma(x_i) b_i , S_i \rangle^{\,i \in 1..n}$	store
$I ::= id_i \rightarrow v_i^{\,i \in 1..n}$	store de identificaciones
$S :: x_i \rightarrow id_i^{\,i \in 1..n}$	stack
$(\sigma , I) \vdash \langle\rangle$	juicio de stores bien formado
$(\sigma, I) \bullet S \vdash \langle\rangle$	juicio de stack bien formado
$(\sigma, I) \bullet S \vdash a \rightsquigarrow id \bullet (\sigma', I')$	juicio de reducción de términos

La representación v de un objeto es la secuencia finita de locaciones de store σ asociadas a los mensajes del objeto. Al igual que en la semántica operacional del impς-cálculo, en el store σ se almacenan method closures, que son pares con un término que representa el método del mensaje y una pila S. En el store I existe una sola entrada por cada objeto, donde se almacena el resultado v correspondiente al objeto. La entrada id que cada objeto tiene en el store I representa su identidad. Una pila S mapea variables a identificaciones de objeto del store I. De esta manera cada referencia a otro objeto se representa a través de su identificación única en el store I.

Las reglas de reducción de términos se presentan en la Figura 4.

(Store ∅)

$$\frac{}{(\emptyset,\emptyset) \vdash \Diamond}$$

(Store ι) *(l_i, ι_i distintos)*

$$\frac{(\sigma, I) \vdash \Diamond \qquad \iota_i \notin dom(\sigma) \qquad i \in 1..n \qquad id \in dom(I)}{((\sigma,\iota_i \rightarrow <\varsigma(x_i)b_i, S_i>^{i \in 1..n}), (I, id \rightarrow [l_i=\iota_i^{\,i \in 1..n}])) \vdash \Diamond}$$

(Stack ∅)

$$\frac{(\sigma, I) \vdash \Diamond}{(\sigma, I) \bullet S \vdash \Diamond}$$

(Stack x)

$$\frac{(\sigma, I) \bullet S \vdash \Diamond \qquad id \in dom(I) \qquad x \notin dom(S)}{(\sigma, I) \bullet (S, x \rightarrow id) \vdash \Diamond}$$

(Red x)

$$\frac{(\sigma, I) \bullet (S', x \rightarrow id, S'') \vdash \vee}{(\sigma, I) \bullet (S', x \rightarrow id, S'') \vdash x \sim> id \bullet (\sigma,I)}$$

(Red Object) *(l_i, ι_i distintos)*

$$\frac{(\sigma, I) \bullet S \vdash \Diamond \qquad id \notin dom(I) \qquad \iota_i \notin dom(\sigma) \qquad \forall\, i \in 1..n}{(\sigma, I) \bullet S \vdash [l_i= \varsigma(x_i)b_i^{\,i \in 1..n}] \sim> id \bullet ((\sigma, \iota_i \rightarrow <\varsigma(x_i)b_i>^{i \in 1..n}, S), (I, id \rightarrow [l_i=\iota_i^{\,i \in 1..n}]))}$$

(Red Select)

$$\frac{(\sigma, I) \bullet S \vdash a \sim> id' \bullet (\sigma', I') \quad I'(id') = [l_i=\iota_i^{\,i \in 1..n}] \quad \sigma'(\iota_j) = <\varsigma(x_j)b_j> \quad x_j \notin dom(S) \quad j \in 1..n \qquad (\sigma', I') \bullet (S, x_j \rightarrow id') \vdash b_j \sim> id'' \bullet (\sigma'', I'')}{(\sigma, I) \bullet S \vdash a.l_j \sim> id'' \bullet (\sigma'', I'')}$$

(Red Let)

$$\frac{(\sigma, I) \bullet S \vdash a \sim> id' \bullet (\sigma', I') \qquad (\sigma', I') \bullet (S, x \rightarrow id') \vdash b \sim> id'' \bullet (\sigma'', I'')}{(\sigma, I) \bullet S \vdash \text{let } x=a \text{ in } b \sim> id'' \bullet (\sigma'', I'')}$$

(Red Update 1)

$$\frac{(\sigma, I) \bullet S \vdash a \sim> id \bullet (\sigma', I') \qquad I'(id) = [l_i=\iota_i^{\,i \in 1..n}] \quad j \in 1..n \quad \iota_j \in dom(\sigma')}{(\sigma, I) \bullet S \vdash a.l_j \Leftarrow \varsigma(x)b \sim> id \bullet ((\sigma', \iota_i \leftarrow <\varsigma(x)b, S>), I')}$$

(Red Update 2)

$$\frac{(\sigma, I) \bullet S \vdash a \sim> id \bullet (\sigma', I') \qquad I'(id) = [l_i=\iota_i^{\,i \in 1..n}] \quad \iota_{n+1} \notin dom(\sigma')}{(\sigma, I) \bullet S \vdash a.l_{n+1} \Leftarrow \varsigma(x)b \sim> id \bullet ((\sigma', \iota_{n+1} \rightarrow <\varsigma(x)b, S>), (I', id \leftarrow [l_i=\iota_i^{\,i \in 1..n+1}]))}$$

(Red Extend 1)

$$\frac{(\sigma, I) \bullet S \vdash a \sim> id \bullet (\sigma', I') \qquad I'(id) = [l_i=\iota_i^{\,i \in 1..n}] \quad \iota_{n+1} \notin dom(\sigma')}{(\sigma, I) \bullet S \vdash a.l_{n+1} \Leftarrow^{+} \varsigma(x)b \sim> id \bullet ((\sigma', \iota_{n+1} \rightarrow <\varsigma(x)b, S>), (I', id \leftarrow [l_i=\iota_i^{\,i \in 1..n+1}]))}$$

(Red Extend 2)

$$\frac{(\sigma, I) \bullet S \vdash a \sim> id \bullet (\sigma', I') \qquad j \in 1..n}{(\sigma, I) \bullet S \vdash a.l_j \Leftarrow^{+} \varsigma(x)b \sim> id \bullet (\sigma', I')}$$

Figura 4

La semántica de la evaluación es básicamente la misma que en el impς-cálculo, salvo que estas reglas mantienen el store de identificaciones *I*.
Una variable reduce a la identificación de objeto a la que está ligada en la pila *S* donde se la está evaluando ***(Red x)***.
En la reducción de un objeto definido directamente ***(Red Object)***, se evalúa a un valor o resultado de la misma manera que se hace en el impς-cálculo y se almacena ese resultado en una nueva locación del store *I*. Esa locación nueva será la identificación del objeto y el resultado de su evaluación.
En el envío de mensajes ***(Red Select)*** se evalúa el objeto receptor obteniendo su identificación. Luego se usa esa entrada en el store *I* para obtener la representación del objeto y seleccionar el mensaje. De la locación de store σ asociada al mensaje se obtiene el código para el método correspondiente, y luego se lo evalúa ligando la variable self a la identidad del objeto receptor.
Para evaluar una construcción let ***(Red Let)*** se reduce primero el objeto a obteniendo su identificación *id*. Luego se liga en la pila la variable del let a esa identificación, y se evalúa el objeto *b*.
Como los operadores $\Leftarrow$ y $\Leftarrow^+$ tienen distinto efecto según si el mensaje que referencian existe o no en el objeto, se definen dos reglas para contemplar ambos casos en la evaluación de cada uno de estos operadores.
En la evaluación de un término $a.l_j \Leftarrow \varsigma(x)b$, se evalúa el objeto a obteniendo su identificación *id* para acceder al conjunto de locaciones en σ asociadas a sus mensajes. Si el mensaje l_j pertenece al protocolo de *a* ***(Red Update 1)*** se actualiza su locación ι_j con el nuevo método. Si en cambio el mensaje l_j no pertenece al protocolo de *a* ***(Red Update 2)*** (l_j es l_{n+1}) se aloca una nueva dirección ι_{n+1} en el store σ y se almacena allí el method closure correspondiente al nuevo mensaje. Luego se modifica la representación del objeto a agregando al resultado que está en su entrada *id*, el nuevo mensaje y su locación asociada ι_{n+1}.
La evaluación del término $a.l_j \Leftarrow^+ \varsigma(x)b$ cuando el mensaje l_j no pertenece al protocolo de a ***(Red Extend 2)***, es exactamente igual que la evaluación del término $a.l_j \Leftarrow \varsigma(x)b$ con la regla ***(Red Update 2)***. Si el mensaje l_j ya pertenece al protocolo de *a*, entonces la evaluación de la extensión con $\Leftarrow^+$ no tiene ningún efecto ***(Red Extend 1)***.
Si bien la definición de extensión y modificación de objetos usando de esta manera los operadores $\Leftarrow$ y $\Leftarrow^+$ puede parecer extraña en principio, se podrá comprender mejor su sentido en los capítulos siguientes al representar los distintos esquemas de sharing.
Al extender un objeto se aloca en el store σ el method closure correspondiente al nuevo mensaje y se modifica el valor de su entrada *id* en el store *I*. El valor se extiende agregándole el nuevo mensaje y la locación en el store σ donde se almacenó su method closure. Así, se extiende la representación del objeto y se conserva su identidad id. En adelante, todo objeto que referencie al objeto extendido podrá percibir los cambios en su protocolo.

Una propiedad que se debe probar sobre la semántica del impEς-cálculo es que las identificaciones de objetos se manejan consistentemente. Es decir, que las reglas de reducción usan y mantienen las entradas *id* en el store de identificaciones *I* en forma consistente. Para ello se debe constatar que el maping *I* es una función inyectiva.
Para ver que *I* es función se debe verificar que toda identificación pertenece a un objeto, y que pertenece a uno y sólo un objeto. Se verifica que toda identificación pertenece a un objeto porque la única regla que aloca nuevas entradas *id* en el store *I* ***(Red Object)***, siempre almacena en la entrada *id* la representación del objeto reducido; además ninguna regla elimina contenidos del store *I*, por lo tanto toda identificación *id* tendrá un objeto asociado. Se cumple que toda identificación corresponde a sólo un objeto porque cuando se asocian identificaciones a objetos

(la regla ***(Red Object)*** es la única que lo hace) se usan locaciones nuevas en el store *I*, por lo tanto una *id* no puede nunca estar asociada a dos objetos distintos.
Para probar la inyectividad de *I* se debe verificar que a identificaciones diferentes corresponden objetos diferentes es decir, que todo objeto tiene una única identificación. Las reglas de reducción cumplen con esta condición, ya que la única que asigna identificaciones a objetos es ***(Red Object)*** y lo hace al alocar un objeto nuevo en el store σ usando locaciones nuevas del store σ, por lo tanto no genera duplicación de identificaciones. Las otras reglas que afectan al store *I*, ***(Red Update 2)*** y ***(Red Extend 1)***, no crean nuevas identificaciones para objetos, sólo modifican el valor *v* contenido en una de ellas. Es decir, no se genera duplicación de identificaciones porque sólo se generan nuevas identificaciones al alocar nuevas direcciones en el store σ.

Ahora veamos cómo la evaluación del ejemplo de extensión del punto-movible presentado anteriormente, tiene la semántica deseada en el impEς-cálculo.

```
. . .
let punto = [ x = ς(self1 ) 5,
              y = ς(self2 ) 6,
              izquierda = ς(self3 ) let valx = (self3.x)  in  (self3 .x ⇐ ς(self1) valx - 1),
              derecha = ς(self4 ) let valx = (self4.x)  in  (self4 .x ⇐ ς(self1) valx + 1),
              arriba = ς(self5 ) let valy = (self5.y)  in  (self5 .y ⇐ ς(self2) valy + 1),
              abajo = ς(self6 ) let valy = (self6.y)  in  (self6 .y ⇐ ς(self2) valy - 1),
              invertir = ς(self7 ) let a = clone(self7) in (self7 .x ⇐ a.y ).y⇐ a.x   ]
   in let dibujo  = [unPixel = ς(self) punto, . . .]
   in let a  =  punto.color⇐+ ς(self) azul
   in unBloque
. . .
```

Suponiendo que el objeto *punto* reduce a la identificación id_{punto}, cuyo contenido es el valor

$$v_{punto} \equiv [x = \iota_1,\ y = \iota_2,\ izquierda = \iota_3,\ derecha = \iota_4,\ arriba = \iota_5,\ abajo = \iota_6,\ invertir = \iota_7]$$

donde ι_1 , ... , ι_7 son locaciones de store σ, al evaluar el objeto *dibujo* se almacenará en su representación una referencia a *punto* a través de id_{punto}. Luego, al extender el objeto *punto* para que tenga color, se aplicará la regla ***(Red Extend 1)*** que modificará el contenido de la entrada id_{punto} cambiándolo por el resultado

$$v_{puntoExt} \equiv [x = \iota_1,\ y = \iota_2,\ izquierda = \iota_3,\ derecha = \iota_4,\ arriba = \iota_5,\ abajo = \iota_6,\ invertir = \iota_7,\ color = \iota_8]$$

De esta manera, si dentro de *unBloque* se envía el mensaje

dibujo.unPixel.color

se reducirá el término *dibujo.unPixel* usando ***(Red Select)***, y se obtendrá la identificación id_{punto}. Luego se evaluará el método del mensaje *color*, usando $v_{puntoExt}$ que es el contenido de la entrada id_{punto} en el store de identificaciones. Observamos además que da lo mismo obtener el color de punto a través de

a.color

ya que la referencia usada aquí (id_{punto}) es la misma que para el término anterior. Y lo mismo ocurre con toda otra referencia a *punto*. Así, cualquier modificación o extensión al objeto *punto* es percibida por todo otro objeto que lo conozca, manteniéndose la integridad referencial.

Sintaxis alternativa

La construcción

$$[m_i = \varsigma(x_i)b_i{}^{i \in 1..n}]$$

sugerida por Abadi y Cardelli para introducir nuevos objetos en el impς-cálculo es demasiado compleja. Define múltiples mensajes (y métodos asociados) de una sola vez, y además establece que el orden entre esos mensajes es indistinto. Es decir que si dos objetos introducidos con este constructor tienen los mismos mensajes pero en orden diferente, son el mismo objeto. Esto puede traer complicaciones al momento de implementar el cálculo.
En el impEς-cálculo, como tiene extensión de objetos, pueden definirse constructores más simples para objetos. Una sintaxis alternativa para este cálculo, al estilo del λ-cálculo de objetos de [FHM94], es

o,b ::=

$[\,]$	Objeto vacío
x	Variable
$o.m$	Envío de mensaje m al objeto o
$o.m \Leftarrow \varsigma(x)b$	Modificación del método asociado al mensaje m
$o.m \Leftarrow^{+} \varsigma(x)b$	Extensión del objeto o con mensaje m
$let\ x = o\ in\ b$	Definición local

Todo objeto que en el impς-cálculo se definía con el constructor *[...]* se puede definir en el impEς-cálculo usando las construcciones $[\,]$ y $o.m \Leftarrow^{+} \varsigma(x)\,b$. Por ejemplo el objeto

$$[m_i = \varsigma(x_i)b_i{}^{i \in 1..n}]$$

puede construirse como

$$(...((\,[\,].m_1 \Leftarrow^{+} \varsigma(x_1)b_1\,).m_2 \Leftarrow^{+} \varsigma(x_2)b_2\,)\ldots\,).m_n \Leftarrow^{+} \varsigma(x_n)b_n$$

Debe observarse que pueden aparcer problemas al momento de definir objetos con métodos mutuamente recursivos. Supongamos que se quiere definir un objeto con un mensaje m_1 que usa al mensaje m_2 del mismo objeto, y que el mensaje m_2 también usa a m_1:

$$[\ m_1 = \varsigma(self_1)\ldots self_1.m_2\ldots,$$
$$\ m_2 = \varsigma(self_2)\ldots self_2.m_1\ldots\ ]$$

La manera de crear este objeto con los constructores propuestos es

$$([\ ].m_1 \Leftarrow^{+} \varsigma(self_1)...self_1.m_2...)\ .\ m_2 \Leftarrow^{+} \varsigma(self_2)...self_2.m_1...$$

(o equivalente invirtiendo el orden de m_1 y m_2). Pero al agregar el primer mensaje, su método asociado referencia al otro mensaje, que todavía no existe en el objeto. Es decir, se producen términos intermedios "incoherentes" según la definición de [Fisher98].
Para salvar este problema se puede usar un truco que consiste en agregar el primer mensaje con un método asociado dummy, luego extender con el segundo mensaje, y luego modificar el primer mensaje con el método original:

$$((\ [\ ].m_1 \Leftarrow^{+} \varsigma(self_1)[\ ]\ )\ .\ m_2 \Leftarrow^{+} \varsigma(self_2)...self_2.m_1...)\ .\ m_1 \Leftarrow \varsigma(self_1)...self_1.m_2...$$

Representación de funciones

Como el impEς-cálculo no contiene una primitiva *clone* las funciones deben representarse de manera diferente a la presentada en el Capítulo 5, donde se utiliza *clone* en para la aplicación de funciones. En esta sección mostramos una representación de funciones utilizando sólo las primitivas de nuestro cálculo.

Una abstracción $\lambda(x)b$ se representa en el impEς-cálculo como un objeto con un mensaje *alocate* que devuelve un objeto con mensajes *arg* y *val*, igual al que se usa para representar funciones en el impς-cálculo. Así, la traducción de una abstracción $\lambda(x)b$ es

$$\begin{aligned} &<< \lambda(x)b >> \ = \ [alocate = \varsigma(x)\ [\ arg = \varsigma(y)\ [\ ], \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad val = \varsigma(z) << b >> \{z \leftarrow z.arg\} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad] \\ &\qquad\qquad\qquad\qquad] \end{aligned}$$

Al aplicar una abstracción se usa el mensaje *alocate* para obtener una copia del objeto con mensajes *arg* y *val* correspondiente a la abstracción. Luego se actualiza el argumento *arg* y se invoca a *val* igual que en la representación de funciones en el impς-cálculo. Así, la traducción de una aplicación de la forma $f(a)$ es

$$\begin{aligned} &<< f(a) >> \ = \ let\ x = << a >>\ in \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (<< f >>.alocate.arg \Leftarrow \varsigma(s)\ x\ ).val \end{aligned}$$

Aquí el mensaje *alocate* cumple el rol de *clone*, generando nuevas copias del objeto con mensajes *arg* y *val* para evitar problemas al invocar la función con distintos argumentos.

Además debe observarse que la traducción de la aplicación evalúa primero el argumento a, obteniendo una referencia x, que es la que efectivamente se usa después para actualizar el argumento *arg*. De esta manera la aplicación tiene pasaje de parámetros por valor, ya que se evalúa el argumento una sola vez y luego se lo usa dentro del cuerpo de la función tantas veces como aparezca.

Capítulo 7

Sharing para Objetos

7. Sharing para Objetos

7.1 Asterix: construcciones de sharing por objetos

En esta sección definiremos Asterix, un pequeño lenguaje de usuario que tiene primitivas para expresar relaciones de sharing entre objetos. Todas las construcciones presentadas en esta sección pueden ser representadas usando el cálculo con extensiones impEς-cálculo, como veremos en la próxima sección.

Asterix tiene primitivas para representar todas las posibles organizaciones de sharing por objetos presentadas en el capítulo III, y no tiene primitivas para definir sharing para grupos de objetos (no existe el concepto de clase).

Igual que en el cálculo, no se hace diferencia entre métodos y variables de instancia; las variables de instancia se representan como mensajes cuyos métodos retornan el valor de la variable.

Los objetos son definidos enumerando los mensajes a los que pueden responder y la respuesta a cada uno de esos mensajes. Los mensajes en Asterix pueden tener argumentos.

La construcción básica de creación de objetos es

```
object
      message-name1  (arg1,..arg_j1) x1,..x_k1 method1 end
      message-name2  (arg2,..arg_j2) x1,..x_k2 method2 end
      ...
      message-namen  (arg1,..arg_jn) x1,..x_kn methodn end
end
```

que permite crear un objeto que puede responder a cada mensaje $\texttt{message-name}_i$ con argumentos $\texttt{arg}_1,..\texttt{arg}_{j1}$ dando $\texttt{method}_i$ como respuesta. Los identificadores $\texttt{xi},..\texttt{x}_{ki}$ representan las variables locales usadas en el método $\texttt{method}_i$.

Los métodos pueden ser objetos definidos de esta misma forma, pueden expresar envío de mensajes, asignaciones, modificaciones de otros métodos, o la evaluación de estas cosas en secuencia.

Esta construcción básica de creación en Asterix esta enriquecida con construcciones para especificar delegación implícita, delegación explícita, embedding implícito y embedding explícito.

Un objeto que delega implícitamente en otro se representa en Asterix con la cláusula **parent,**

```
object
      parent o1
      message-name1  (arg1,..arg_j1) x1,..x_k1 method1 end
      ...
      message-namen  (arg1,..arg_jn) x1,..x_kn methodn end
end
```

Esto permite definir un objeto que delega en otro objeto, que llamamos parent, todos aquellos mensajes para los que no define su propio comportamiento. En este caso el objeto o podrá responder a todos los mensajes definidos por el objeto o_1, usando el comportamiento que éste define. Los mensajes $\texttt{message-name}_1,...,\texttt{message-name}_n$ describen el comportamiento

adicional del objeto, o redefinen comportamiento delegado (si alguno de los mensajes $message\text{-}name_i$ pertenece al protocolo de o_1).

Si un objeto delega implícitamente en otro puede enviarle mensajes referenciándolo, dentro de sus métodos, como **super**.

Asterix provee delegación explícita mediante la cláusula **delegate**:

```
object
      message-name 1 delegate in o1
      message-name 2 (arg2,..argj2) x1,..xk2 method2 end
      ...
      message-name n (arg1,..argjn) x1,..xkn methodn end
end
```

Esta construcción crea un objeto que delega en el objeto o_1 el mensaje $message\text{-}name_1$. Cuando el objeto borrower reciba el mensaje $message_1$ responderá con el comportamiento definido en el objeto o_1 para ese mensaje.

Hay construcciones similares a las presentadas para permitir incorporar comportamiento definido por otro objeto. Embedding implícito se expresa en Asterix con la primitiva **clone**:

```
object
      clone o1
      message-name 1 (arg1,..argj1) x1,..xk1 method1 end
      ...
      message-name n (arg1,..argjn) x1,..xkn methodn end
end
```

El nuevo objeto incorpora todo el comportamiento del donor, y además tiene definido comportamiento individual (que pueden ser nuevos mensajes o redefiniciones de respuestas a algunos mensajes del objeto o_1). A partir de este momento, cuando el borrower reciba un mensaje no interactuará con el donor, sino que responderá en forma autónoma.

Finalmente, también hay una construcción para especificar embedding explícito,

```
object
      message-name 1 embed from o1
      message-name 2 (arg2,..argj2) x1,..xk2 method2 end
      ...
      message-name n (arg1,..argjn) x1,..xkn methodn end
end
```

En este caso se crea un objeto que puede responder a los mensajes $message\text{-}name_i$, $1 \leq i \leq n$, donde la respuesta al mensaje $message\text{-}name_1$ es una copia de la respuesta del objeto o_1 a ese mensaje.

Estas construcciones de sharing pueden combinarse, un objeto puede delegar explícitamente mensajes en varios diferentes donors, y a la vez embeber explícitamente otros mensajes de distintos donors; un mismo objeto también puede tener varios donors explícitos y un donor implícito. La única restricción es que cada objeto Asterix puede tener a lo sumo un donor implícito, ya sea por delegación o por embedding.

Modificación

El comportamiento de un objeto en Asterix puede modificarse cambiando la respuesta a un mensaje o cambiando el parent de un objeto borrower.

Para que ningún objeto pueda cambiar el comportamiento de otro violando el encapsulamiento, Asterix impone una restricción sobre las modificaciones: sólo se pueden hacer modificaciones sobre la variable self, dentro de un método del objeto que está siendo modificado.

Cualquiera haya sido la respuesta a un mensaje esta puede modificarse, y la nueva respuesta puede ser comportamiento propio definido o puede ser delegada o embebida de otro objeto. La modificación en Asterix del mensaje `message-name` de un objeto se expresa así:

```
self message-name := ( (arg1,..argj1) x1,..xk1 method1 end |
                       delegate in o1 |
                       embed from o1 |
                       embed from parent |
                       delegate in parent )
```

embed from `parent` y **delegate in** `parent` pueden usarse si el objeto delega implícitamente en otro, es decir, tiene un parent, y significan que la respuesta al mensaje se delega o embebe del parent.

Si un objeto delega implícitamente en otro puede cambiar de donor con la construcción

`self` **ChangeParent:** o_1

con la condición de que el nuevo parent entienda los mismos mensajes del donor original.

La sintaxis completa de Asterix aparece en la Figura 5.

7.2 Semántica Formal de Asterix

En esta sección presentaremos el modelo que permite representar todos los esquemas de sharing por objeto en el impEς-cálculo. Para ello mostraremos cómo traducir cada una de las construcciones definidas en Asterix a términos en el cálculo. Primero describiremos la forma general de la representación de los objetos de Asterix en el cálculo, y luego cómo se representa cada esquema de sharing dentro de esta forma general. La traducción de Asterix al cálculo se define como una función de términos de Asterix en términos del impEς-cálculo. La definición completa de la función de traducción se encuentra en el Apéndice

Representación de los objetos de Asterix

El primer requerimiento para poder representar sharing por objeto es que el comportamiento definido por cualquier objeto pueda ser reusado en otros objetos. El comportamiento compartido se debe ejecutar en el contexto del receptor original del mensaje, es decir, que toda referencia a self en el método reusado se refiere al receptor original, y no al objeto que define el comportamiento.

En el cálculo no se permite extraer un método de un objeto; un método sólo puede ser evaluado cuando se recibe el mensaje correspondiente. En ese momento la variable ligada por ς , se asocia al objeto que define el método. Por lo tanto, para poder compartir es necesario realizar construcciones que permitan obtener el comportamiento de un objeto para un mensaje dado, y ejecutarlo asociando las referencias a self al receptor original del mensaje.

La representación que proponemos para permitir compartir comportamiento consiste en representar cada mensaje que un objeto puede recibir de sus clientes mediante dos mensajes: un mensaje de especificación y un mensaje de implementación. El mensaje de especificación lleva el nombre del mensaje original, y es el que se recibirá de los clientes. El mensaje de implementación contiene el comportamiento efectivo del mensaje original y tiene un parámetro que permite instanciar el receptor original del mensaje. El método de especificación invoca al mensaje de implementación, instanciando el receptor con la variable self (la variable ligada por ς). El método de implementación es similar a los premétodos presentados en la sección 5.2: toda vez que el método original referencia a la variable ligada por ς, el método de implementación referencia al parámetro que representa al receptor. Los métodos de implementación se diferencian de los premétodos en las operaciones de modificación: en el método de implementación las modificaciones de los métodos asociados a los mensajes se realizan sobre el mensaje de implementación correspondiente al mensaje que se desea modificar. El resultado de un mensaje de implementación es un objeto función, que contiene el comportamiento efectivo correspondiente al método que le dio origen, y que puede ejecutarse en el contexto de cualquier objeto que se use como parámetro.

Un objeto simple definido en Asterix como

```
object
      message1 (arg1,..arg_j1) method1 end
      message2 (arg2,..arg_j2) method2 end
      ...
      messagen (arg1,..arg_jn) methodn end
end
```

estará representado en el cálculo por

$[$ $message_1$: $\varsigma(x_1)\, x_1.*message_1(x_1)$,
$message_2$: $\varsigma(x_2)\, x_2.*message_2(x_2)$,
...
$message_n$: $\varsigma(x_n)\, x_n.*message_n(x_n)$,
$*message_1$: $\varsigma(z_1)\, \lambda(rec)\, \lambda(arg_1)\, \lambda(arg_2)...\lambda(arg_{j1})$
$method_1$-traducido{self.m ←- rec.m , self.m⇐ ←rec.*m⇐ },
$*message_2$: ...
...
$*message_n$:....]

Con respecto a los mensajes $m_1..m_n$ esta representación se comporta igual que una representación directa (sin premétodos, donde se usa la variable ligada por ς para las referencias a self).

La división de los mensajes en especificación e implementación es la clave de la representación que permite representar los distintos esquemas de sharing. El método de especificación es igual para todos los esquemas y el método de implementación cambia en la representación de cada esquema.

Representación de delegación

Cuando un objeto delega un cierto mensaje en otro objeto, al recibir ese mensaje se debe obtener del donor el comportamiento efectivo para responder a ese mensaje.

En el cálculo los objetos sólo pueden responder a mensajes que pertenezcan a su definición. Por lo tanto, para simular la delegación en el cálculo, aquellos mensajes que se delegan deben estar incluidos en la representación del objeto. Siguiendo con el esquema de representación definido, el objeto tendrá un mensaje de especificación y uno de implementación para cada mensaje delegado. La parte de especificación será igual que en todos los objetos. La parte de implementación obtendrá el comportamiento efectivo para el mensaje enviando el mensaje de implementación al objeto donor, utilizando el receptor recibido como parámetro.

Un objeto que delega el mensaje $message_1$ en un objeto o_2 estará definido en Asterix como

```
object
      message1 delegate in o2 end
      ...
end
```

y estará representado en el cálculo por

$[$ $message_1$: $\varsigma(x_1)\, x_1.*message_1(x_1)$,
...
$*message_1$: $\varsigma(z_1)\, \lambda(rec)\, o_2.*message_1(rec)$,
...]

El comportamiento del donor se obtiene cuando se recibe el mensaje delegado (en este caso $message_1$) , por lo tanto, si el donor cambia su comportamiento, este cambio afectará al objeto que delega en él, que exhibirá este nuevo comportamiento cuando reciba el mensaje delegado.

Podría suceder que el donor designado o_2, a su vez delegue el mensaje compartido en un tercer objeto o_3. Cuando el borrower envíe a o_2 el mensaje $*message_1$, éste a su vez enviará el mismo mensaje a o_3, y así se seguirá la cadena de delegación hasta llegar al objeto que implementa efectivamente el comportamiento para $message_1$.

La delegación representada anteriormente es *explícita*, ya que se designa explícitamente un donor para el mensaje $\texttt{message}_1$.

En la *delegación implícita* un objeto designa un objeto parent en el cual se delegan todos aquellos mensajes para los cuales el objeto no define un comportamiento propio. Por lo tanto, este objeto podrá responder al menos a todos los mensajes de su parent. Como ya discutimos, los mensajes delegados deben incluirse en el objeto que delega; para simular delegación implícita se deben incluir en el objeto borrower todos los mensajes de su parent. La implementación de estos mensajes es similar a la descripta para delegación explícita. En este caso, el borrower conocerá quien es su parent y obtendrá de él el comportamiento delegado.

Un objeto que tiene como parent un objeto o_2 que entiende los mensajes $message_1$,...,$message_n$ se escribe en Asterix

```
object
      parent o2
end
```

y se podría representar como

$$
\begin{aligned}
&[\ message_1\colon \varsigma(x_1)\ x_1.*message_1(x_1),\\
&\quad \ldots\\
&\quad message_n\colon \varsigma(x_n)\ x_n.*message_n(x_n),\\
&\quad parent\colon \varsigma(x)\ o_2\ ,\\
&\quad *message_1\colon \varsigma(z_1)\ \lambda(rec)\ z_1.parent.*message_1(rec),\\
&\quad \ldots\\
&\quad *message_n\colon \varsigma(z_n)\ \lambda(rec)\ z_n.parent.*message_n(rec)]
\end{aligned}
$$

Ahora veamos cómo incorporar todos los mensajes del parent en el objeto borrower. Si se conoce el protocolo del objeto parent en el momento de realizar la traducción se puede escribir fácilmente la representación del borrower incluyendo los mensajes de especificación e implementación correspondientes. Pero esta solución es muy restrictiva, ya que no permite crear dinámicamente objetos que deleguen en cualquier otro. En el cálculo no hay forma de manipular el conjunto de mensajes de un objeto desde el lenguaje después de que el objeto se creó. El único momento en que se puede conocer el protocolo completo de un objeto es cuando se define el objeto con la construcción *[...]*. Por lo tanto, al definir un objeto de esta manera, instalaremos en él un mensaje llamado *delegate* que permitirá crear objetos con su mismo protocolo y que deleguen en el creador todo su comportamiento.

El nuevo objeto creado también debe tener un mensaje *delegate*, que hace lo mismo que el del creador (pero creando objetos que delegan en el nuevo). Intentar escribir este mensaje en forma directa llevaría a un término infinito:

$$
\begin{aligned}
&[\ message_i\colon \varsigma(x_i)\ x_i.*message_i(x_i),^{i\in 1..n}\\
&\quad \ldots\\
&\quad *message_i\colon \varsigma(z_i)\ \lambda(rec)\ \lambda(arg_1)\ \lambda(arg_2)\ldots\lambda(arg_{ji})\ method_i\ ,\\
&\quad delegate\colon \varsigma(x)\quad [message_i\colon \varsigma(x_i)\ x_i.*message_i(x_i),^{i\in 1..n}\\
&\qquad\qquad \ldots\\
&\qquad\qquad parent\colon \varsigma(z)x,\\
&\qquad\qquad *message_i\colon \varsigma(z_i)\ \lambda(rec)\ z_i.parent.*message_i(z_i)\ ,\\
&\qquad\qquad delegate\colon \varsigma(z)\ [message_i\colon \varsigma(x_i)\ x_i.*message_i(x_i),^{i\in 1..n}\\
&\qquad\qquad parent\colon \varsigma(w)z,\\
&\qquad\qquad *message_i\colon \varsigma(z_i)\ \lambda(rec)\ z_i.parent.*message_i\ (rec)\ ,\\
&\qquad\qquad delegate\colon \varsigma(v)\ [.....\\
&\qquad\qquad\qquad\qquad delegate\colon.....
\end{aligned}
$$

Para solucionar este problema, utilizamos la técnica de premétodos en la representación del mensaje delegate. De este modo, el objeto anterior se representa como:

$[\ message_i: \varsigma(x_i)\ x_i.*message_i(x_i),^{i\in 1..n}$
$\ldots$
$*message_i: \varsigma(z_i)\ \lambda(rec)\ \lambda(arg_1)\ \lambda(arg_2)\ldots\lambda(arg_{ji})\ method_i,$
$delegate: \varsigma(x)\ x.*delegate(x),$
$*delegate: \varsigma(x)\ \lambda(rec)\ let\ d = rec.*delegate\ in$
$[\quad message_i: \varsigma(x_i)\ x_i.*message_i(x_i),^{i\in 1..n}$
$\ldots$
$parent: \varsigma(z)rec,$
$*message_i: \varsigma(z_i)\ \lambda(rec)\ z_i.parent.*message_i(rec),$
$delegate: \varsigma(x)\ x.*delegate(x),$
$*delegate: \varsigma(x)\ d]$
$]$

De este modo podemos obtener una copia del premétodo **delegate* del creador (*let d = rec.*delegate...*) e incluirla en el nuevo objeto creado. El parent del nuevo objeto es el receptor del mensaje delegate, que es el parámetro *rec* de **delegate*. Notar que el método para **delegate* se embebe en el objeto creado.

Con esta construcción, el objeto que en Asterix es

```
object
    parent o2
end
```

se representa con

$o_2.delegate$

Esta representación permite crear objetos que delegan todo su comportamiento en su parent. Más adelante mostraremos cómo representar objetos que además de delegar implícitamente en otro, definen comportamiento propio.

Representación de embedding

Ahora analizaremos como representar embedding explícito e implícito en el marco de la forma general propuesta. Al compartir comportamiento mediante embedding, el objeto borrower incorpora comportamiento definido en otro objeto donor, y en lo sucesivo responderá a los mensajes embebidos sin interactuar con el donor. El método para cada mensaje embebido será una copia del método correspondiente del donor. Como ya discutimos, el cálculo no permite extraer un método de un objeto. Sin embargo, la representación con especificación e implementación nos permite obtener el premétodo correspondiente mediante el mensaje de implementación, y así copiarlo e incorporarlo en el objeto borrower.

Un objeto que incorpora explícitamente el mensaje `messemb` de un objeto o_2, se escribe en Asterix

```
object
    message_i (arg_i,..arg_ji) method_i end
    messemb embed from o_2 end
end
```

y se representa con

$$
\begin{array}{l}
\textit{let mcode=}o_2\textit{ .*messemb in} \\
\quad [\quad \textit{message}_i\textit{: } \varsigma(x_i)\ x_i.\textit{*message}_i(x_i)^{i\in 1..n}\textit{,} \\
\qquad \textit{messemb: } \varsigma(x)\ x.\textit{*messemb (x),} \\
\qquad \textit{*message}_i\textit{: } \varsigma(z_i)\ \lambda(rec)\ \lambda(arg_1)\ \lambda(arg_2)...\lambda(arg_{j\,i})\ \textit{method}_i\ ^{i\in 1..n}\textit{,} \\
\qquad \textit{*messemb: } \varsigma(z)\ \lambda(rec).mcode(rec)\ ]
\end{array}
$$

La construcción *let mcode=o2.* messemb in...* genera una copia del premétodo para messemb del donor y la incorpora como implementación de messemb en el borrower.

El embedding implícito designa un objeto donor e incorpora de éste todo el comportamiento. El efecto de este esquema es la creación de una copia del donor. La creación de copias de un objeto plantea los mismos problemas que se discutieron para delegación implícita con respecto a la incorporación del protocolo del donor. La solución es similar: agregamos a cada objeto un mensaje clone definido en estilo de premétodos, parecido al mensaje delegate utilizado para representar delegación implícita. Ambas representaciones están inspiradas por la implementación de clone dada en [AC96].

El objeto creado por el mensaje clone tiene los mismo mensajes que el receptor, los mensajes de especificación son iguales que los de las construcciones anteriores y los métodos de implementación son copias de los métodos de implementación del receptor (incluso el *clone).

Un objeto cuyos mensajes son $\texttt{message}_1, \ldots, \texttt{message}_n$ se representa como:

$$
\begin{array}{l}
[\ \textit{message}_i\textit{: } \varsigma(x_i)\ x_i.\textit{*message}_i(x_i)\textit{,}^{i\in 1..n} \\
\quad ... \\
\quad \textit{*message}_i\textit{: } \varsigma(z_i)\ \lambda(rec)\ \lambda(arg_1)\ \lambda(arg_2)...\lambda(arg_{j\,i})\ \textit{method}_i\textit{,} \\
\quad \textit{clone: } \varsigma(x)\ x.\textit{*clone(x),} \\
\quad \textit{*clone: } \varsigma(x)\ \lambda(rec)\ \textit{let c=rec.*clone in} \\
\qquad \textit{let } m_1\textit{code=rec.*message}_1\textit{ in} \\
\qquad ... \\
\qquad \textit{let } m_n\textit{code=rec.*message}_n\textit{ in} \\
\qquad [\textit{message}_i\textit{: } \varsigma(x_i)\ x_i.\textit{*message}_i(x_i)\textit{,}^{i\in 1..n} \\
\qquad ... \\
\qquad \textit{*message}_i\textit{: } \varsigma(z_i)\ \lambda(rec)\ m_1\textit{code(rec),} \\
\qquad \textit{clone: } \varsigma(x)\ x.\textit{*clone(x),} \\
\qquad \textit{*clone: } \varsigma(x)\ c] \\
]
\end{array}
$$

Cada objeto de la representación tendrá los mensajes definidos para implementar delegación implícita (delegate y *delegate) y embedding implícito (clone y *clone). De esta manera pueden crear objetos que compartan con ellos de cualquiera de las dos maneras.

Sharing dinámico

Las construcciones presentadas hasta ahora permiten establecer relaciones de sharing. La definición de Asterix permite también cambiar las relaciones de sharing: cambiando el parent de un objeto o la forma de compartir un cierto mensaje. Cualquiera sea la forma en que se respondía a un mensaje, se puede comenzar a delegar, embeber o definir comportamiento propio para ese mensaje. Recordemos que la modificación de las relaciones de sharing sólo puede ser realizada por el propio objeto. Estos cambios se implementan mediante actualización de objetos. La actualización se realiza sobre la variable que representa a self (ligada por ς).

El cambio de parent se implementa en la representación actualizando el método parent para que retorne el nuevo parent. Este nuevo parent debe tener los mismo mensajes que el reemplazado, ya que si faltan mensajes se producirá un error al intentar recuperar el comportamiento para los mensajes delegados y si tiene más, existirán en el parent mensajes cuyo comportamiento no se reúsa en el borrower.

El cambio

self changeParent: o_2

se representa como (suponiendo que x es la variable que representa a self)

$x.parent \Leftarrow \varsigma(x)\, o_2$

Los demás cambios se implementan actualizando el premétodo correspondiente al mensaje cuya relación cambia.

Comenzar a delegar un mensaje se expresa en Asterix

self m_1 **:=delegate in** o_1

y se traduce como

$x.*m_1 \Leftarrow \varsigma(z)\lambda(rec)\, o_1.*m_1(rec)$

Incorporar un mensaje, en Asterix se escribe

self m_1 **:=embed from** o_1

y se traduce como

let $m_1code=o_1.*m_1$ *in*

$x.*m_1 \Leftarrow \varsigma(z)\ \lambda(rec)\ m_1code(rec)$

Para traducir **delegate in parent** y **embed from parent** se usará *x.parent* en lugar de o_1.

Especialización

Las representaciones de sharing implícito que hemos descripto permiten generar objetos que tienen exactamente el mismo conjunto de mensajes que su donor implícito. Sin embargo, las construcciones propuestas en el lenguaje Asterix permiten que un objeto además de designar un donor implícito pueda definir comportamiento propio. Este comportamiento propio pueden ser nuevos mensajes o definiciones propias para mensajes que están en el padre. La representación de esta situación motivó la definición del cálculo con reglas para extensión de objetos presentado en el capítulo 6.

La redefinición de un método del donor, se resuelve simplemente actualizando el método de implementación con el premétodo correspondiente al nuevo comportamiento.

Un objeto que tiene un donor implícito y define además nuevos mensajes se podría crear definiendo un término con los mensajes del donor y los nuevos. Pero ya vimos que definir un término con los mensajes del donor requiere conocer estáticamente su conjunto de mensajes, resultando poco flexible. Es por esto que se resolvió que los objetos que delegan implícitamente en otro, sean creados por el donor mismo (mediante el mensaje delegate o clone, según se trate de delegación o embedding). Para definir nuevos mensajes para estos objetos creados por el donor es necesario agregar mensajes a un objeto ya creado. El impς-cálculo no permite extender los objetos una vez creados, por lo tanto fue necesario definir el cálculo con extensiones para contemplar esta situación.

Al definir comportamiento propio para un cierto mensaje no es posible saber estáticamente si el mensaje está o no definido en el donor (ya que no se requiere conocer estáticamente su protocolo), por lo cual la representación de la especialización debe ser capaz de contemplar

ambas situaciones. Ya vimos que la redefinición de un mensaje del parent se puede resolver utilizando el operador de update $\Leftarrow$, como está definido en el impς-cálculo. Para contemplar a la vez que el mensaje pertenezca o no al parent, al definir la extensión en impEς-cálculo decidimos utilizar el mismo operador para agregar nuevos mensajes, obteniendo el operador de extensión no-conservativa que se describe en el Capítulo 6.

Cuando en Asterix se define comportamiento propio para un mensaje, en la representación se agregan o actualizan los mensajes de especificación e implementación correspondientes. En el caso de una redefinición no sería necesario actualizar el mensaje de especificación, pero hay que hacerlo para tener un tratamiento uniforme, ya que no se puede saber si el mensaje es nuevo o no.

Utilizando el nuevo operador, el objeto que en Asterix se define como

```
object
      parent o2
      myMessage(arg1,..argj) myMethod end
end
```

se representa como

*(o_2 .delegate .myMessage $\Leftarrow$ ς(x) x.*myMessage(x))*
*.*myMessage$\Leftarrow$ ς(x) λ(rec) λ(arg_1)..λ(arg_j)myMethod-traducido*

Análogamente, el objeto definido como

```
object
      clone o2
      myMessage(arg1,..argj) myMethod end
end
```

se representa como

*(o_2 .clone .myMessage $\Leftarrow$ ς(x) x.*myMessage(x))*
*.*myMessage$\Leftarrow$ ς(x) λ(rec) λ(arg_1)..λ(arg_j)myMethod-traducido*

Cuando un objeto se extiende con nuevos mensajes, los objetos que posteriormente se creen a partir de él mediante el mensaje clone o delegate deben incluir los nuevos mensajes. Para que esto suceda, al realizar la extensión se debe actualizar la implementación de los mensajes clone y delegate. La nueva implementación ejecutará la implementación vieja, y extenderá el objeto resultado con los nuevos mensajes.

Teniendo en cuenta la modificación de clone y delegate, la representación completa del objeto anterior es

let o=o2.clone in
*let old*d=o.*delegate in*
*let old*c=o.*clone in*
*o .myMessage $\Leftarrow$ ς(x) x.*myMessage(x)*
*.*myMessage$\Leftarrow$ ς(x) λ(rec) λ(arg_1)..λ(arg_j)myMethod-traducido*
*.*delegate$\Leftarrow$ ς (z) λ(rec) old*d(rec).myMessage$\Leftarrow$ ς(x) x.*myMessage(x)*
*.*myMessage$\Leftarrow$ ς(x) λ(rec_1) rec.*myMessage(rec_1)*
*.*clone$\Leftarrow$ ς (z) λ(rec) old*c(rec).myMessage$\Leftarrow$ ς(x) x.*myMessage(x)*
*.*myMessage$\Leftarrow$ ς(x) λ(rec) λ(arg_1)..λ(arg_j)*
myMethod-traducido

La posibilidad de extender objetos incorporada al modelo se puede utilizar en cualquier momento, no sólo al crear objetos. Siempre que se extienda un objeto, se deberá actualizar la implementación de clone y delegate de la forma descripta anteriormente.

La posibilidad de extender cualquier objeto y en cualquier momento tiene algunas consecuencias cuando se la aplica a un objeto que actúa como parent de otros. Según se definió delegación implícita, un objeto delega en su parent todos aquellos mensajes para los cuales no define comportamiento propio. Por lo tanto, si el parent agrega un nuevo mensaje, los objetos que delegan en el también serán capaces de responder a ese mensaje, utilizando el comportamiento definido en el parent. También podría suceder que alguno de los borrowers ya hubiese agregado ese nuevo mensaje, con lo cual la extensión del parent no tendrá ningún efecto sobre él, que continuará respondiendo al mensaje con su propio comportamiento.

A diferencia de lo que sucede con los objetos que delegan implícitamente en un donor cuando éste se extiende, los objetos creados por embedding implícito no se ven afectados por las modificaciones realizadas sobre su donor implícito.

Como ya sabemos, los objetos en el cálculo sólo pueden responder a los mensajes que tienen definidos, por lo tanto para que los borrowers puedan responder al nuevo mensaje agregado al parent se debe agregar este mensaje a cada uno. Para implementar esto, cada objeto mantendrá una lista de los objetos creados a partir de él, y cuando se lo extienda, se propagará la extensión a todos ellos. También se propagará la extensión a los objetos creados a partir de estos últimos. Junto con el agregado de los nuevos mensajes, se debe propagar la modificación a *clone y *delegate correspondiente a este agregado.

Al propagar la extensión se debe tener en cuenta que si el mensaje ya fue agregado en uno de los objetos afectados, se debe conservar el comportamiento definido por éste último. Por esto, no podemos usar el operador $\Leftarrow$ para propagar la extensión, ya que este operador sobreescribe el comportamiento existente y como no es posible saber si el mensaje está definido en el objeto o no, no se puede aplicar la extensión en forma condicional. Para resolver esta situación, agregamos al cálculo el operador de extensión conservativa $\Leftarrow^+$, que agrega un mensaje a un objeto sólo cuando no está definido. Este nuevo operador se utiliza para propagar la extensión de un objeto a todos aquellos objetos que delegan en él, ya sea directa o indirectamente.

Para implementar la propagación, se agregan a la representación de cada objeto, mensajes para mantener y manipular la lista de objetos que delegan implícitamente en él (*children*) y para propagar modificaciones.

Integración de los esquemas

La definición de Asterix permite combinar los distintos esquemas de sharing entre sí. En la sección anterior, se combinaba sharing implícito con comportamiento definido por el objeto. También es posible especializar mensajes mediante delegación o embedding explícito. La combinación de los esquemas se realiza en forma modular, componiendo las construcciones descriptas para los distintos tipos de sharing.

Al definir un objeto en Asterix se puede especializar un conjunto de mensajes, usando diferentes formas de sharing en cada uno y esto se traducirá mediante extensiones sucesivas. Estas extensiones tendrán la forma descripta en el punto anterior (Especialización), pero el cuerpo del mensaje de implementación tendrá la forma correspondiente al tipo de sharing elegido.

A continuación veremos un ejemplo de cómo se realiza la composición de distintos esquemas de sharing. En Asterix es posible definir el siguiente objeto:

object
parent o1
message m1 **delegate in** o2
end

Este objeto tiene como parent al objeto o1, delega el mensaje m1 en el objeto o2. Se traduce al cálculo de la siguiente manera:

$$
\begin{array}{l}
\textit{let } o=o1.clone \textit{ in} \\
\textit{let } old*d=o.*delegate \textit{ in} \\
\textit{let } old*c=o.*clone \textit{ in} \\
\qquad o.m1 \Leftarrow \varsigma(x)\, x.*m1(x) \\
\qquad\quad .*m1 \Leftarrow \varsigma(x)\, \lambda(rec)\, o2.*m1(rec) \\
\qquad\quad .*delegate \Leftarrow \varsigma(z)\, \lambda(rec)\, old*d(rec).m1 \Leftarrow \varsigma(x)\, x.*m1(x) \\
\qquad\qquad\qquad\qquad .*m1 \Leftarrow \varsigma(x)\, \lambda(rec_1)\, o2.*m1(rec_1) \\
\qquad\quad .*clone \Leftarrow \varsigma(z)\, \lambda(rec)\, old*c(rec)\, .m1 \Leftarrow \varsigma(x)\, x.*m1(x) \\
\qquad\qquad\qquad\qquad .*m1 \Leftarrow \varsigma(x)\, \lambda(rec_1)\, o2.*m1(rec_1)
\end{array}
$$

Si además, el objeto anterior embebe el mensaje m2 del objeto m3, la extensión correspondiente al nuevo mensaje se agregará a la traducción anterior. El objeto se define en Asterix de la siguiente manera

object
parent o1
message m1 **delegate in** o2
message m2 **embed from** o3
end

y su traducción completa es

$$
\begin{array}{l}
\textit{let } o= \\
\quad \textit{let } o=o1.delegate \textit{ in} \\
\quad \textit{let } old*d=o.*delegate \textit{ in} \\
\quad \textit{let } old*c=o.*clone \textit{ in} \\
\qquad o.m1 \Leftarrow \varsigma(x)\, x.*m1(x) \\
\qquad\quad .*m1 \Leftarrow \varsigma(x)\, \lambda(rec)\, o2.*m1(rec) \\
\qquad\quad .*delegate \Leftarrow \varsigma(z)\, \lambda(rec)\, old*d(rec).m1 \Leftarrow \varsigma(x)\, x.*m1(x) \\
\qquad\qquad\qquad\qquad .*m1 \Leftarrow \varsigma(x)\, \lambda(rec_1)\, o2.*m1(rec_1) \\
\qquad\quad .*clone \Leftarrow \varsigma(z)\, \lambda(rec)\, old*c(rec)\, .m1 \Leftarrow \varsigma(x)\, x.*m1(x) \\
\qquad\qquad\qquad\qquad .*m1 \Leftarrow \varsigma(x)\, \lambda(rec_1)\, o2.*m1(rec_1) \\
\textit{in} \\
\textit{let } old*d=o.*delegate \textit{ in} \\
\textit{let } old*c=o.*clone \textit{ in} \\
\qquad o.m2 \Leftarrow \varsigma(x)\, x.*m2(x) \\
\qquad\quad .*m2 \Leftarrow \varsigma(x)\, \lambda(rec)\, o3\, .*m2(rec) \\
\qquad\quad .*delegate \Leftarrow \varsigma(z)\, \lambda(rec)\, old*d(rec).m2 \Leftarrow \varsigma(x)\, x.*m2(x) \\
\qquad\qquad\qquad\qquad .*m2 \Leftarrow \varsigma(x)\, \lambda(rec_1)\, o3.*m1(rec_1) \\
\qquad\quad .*clone \Leftarrow \varsigma(z)\, \lambda(rec)\, old*c(rec)\, .m2 \Leftarrow \varsigma(x)\, x.*m2(x) \\
\qquad\qquad\qquad\qquad .*m2 \Leftarrow \varsigma(x)\, \lambda(rec_1)\, o3.*m1(rec_1)
\end{array}
$$

Objeto Top

Todos los objetos de la representación tienen un comportamiento en común, que es el conjunto de mensajes que les permite soportar el sharing implícito (clone, delegate, y los mensajes para propagación de cambios). En la representación, ese comportamiento común está definido en un objeto que llamamos *Top* (se puede ver en la Figura 6, la definición completa está en el Apéndice). Los objetos que no definen un donor implícito, se crean como copias del objeto top (por medio del mensaje top.clone), extendidos con su comportamiento específico en la forma descripta en el apartado de especialización. Los objetos que definen un donor implícito se crean como ya vimos, a partir de éste. De esta manera se forma una jerarquía de objetos relacionados por sharing implícito (embedding y delegación), cuya raíz es top. En definitiva, la creación de un objeto en la representación consistirá en una copia del objeto top, seguida de una sucesión de extensiones dada por las especializaciones definidas en los objetos de la cadena de sharing implícito que va de top a su donor directo.

La definición del objeto top utiliza la representación de los booleanos y una representación de listas (Ver Apéndice). Los mensajes clone y delegate tienen la forma que describimos anteriormente. El objeto también describe el manejo de una lista de los borrowers por delegación implícita del objeto (*children*), es decir, de aquellos objetos a quienes se deben propagar las extensiones del objeto. Esta lista se mantiene por medio de los mensajes *register* y *unregister,* que agregan y borran un objeto de la lista de children, respectivamente. Para implementar el borrado y agregado se usa la pregunta de si un objeto es idéntico a otro; esta pregunta se implementa mediante los mensajes *test* y =*?*.

```
Top ≡
    let ch = Lista.new in
        [test = ς(self) False,
        =? = ς(self) λ(other) let a = other.test ⇐ False
                              let b = self.test ⇐ True in other.test,
     children = ς(self) ch,
     register = ς(self) λ(child) self.children.agregar(child),
     unregister = ς(self) λ(child) self.children.borrar(child),
     propagate = ς(self) λ(f) let me = f(self) in self.children.do(f),
        clone = ς(self) self.*clone(self),
     delegate = ς(self) self.*delegate(self),
     *delegate = ς(self)λ(rec) let d = rec.*delegate in
                               let c = rec.*clone in
                               let ch = Lista.new in
                               let newObj =
                                    [ parent = rec,
                                    delegate = ς(xd) xd.*delegate(xd),
                                    *delegate = ς(xd) d,
                                    clone = ς(xd) xd.*clone(xd),
                                    *clone = ς(xd) c,
                                    test = ς(self) False,
                                    =? = ς(self)λ(other)let a = other.test ⇐ False in
                                                        let b=self.test⇐ True in other.test,
                                    children = ς(self) ch,
                                    register = ς(self) λ(child) self.children.agregar(child),
                                    unregister = ς(self) λ(child) self.children.borrar(child),
                                    propagate = ς(self) λ(f) let me = f(self) in
                                                                    self.children.do(f) ]
                               in
                               let registration = rec.register(newObj) in newObj
     *clone = ς(self)λ(rec) let d = rec.*delegate in
                            let c = rec.*clone in
                            let ch = Lista.new in
                                    [ delegate = ς(xd) xd.*delegate(xd),
                                    *delegate = ς(xd) d,
                                    clone = ς(xd) xd.*clone(xd),
                                    *clone = ς(xd) c,
                                    test = ς(self) False,
                                    =? = ς(self) λ(other)let a = other.test ⇐ False in
                                                        let b = self.test ⇐ True in
                                     other.test,
                                    children = ς(self) ch,
                                    register = ς(self) λ(child) self.children.agregar(child),
                                    unregister = ς(self) λ(child) self.children.borrar(child),
                                    propagate = ς(self) λ(f) let me = f(self) in
                                                                    self.children.do(f) ]
        ]
```

Figura 6

Capítulo 8

Trabajos Relacionados

8. Trabajos Relacionados

En este capítulo comparamos nuestro trabajo con algunos cálculos que definen extensión de objetos y con uno que define operadores de delegación como primitivos.

8.1 Cálculos de la línea de Fisher-Honsell-Mitchel

En la línea de estudio iniciada por Fisher, Honsell y Mitchel con [MHF94] se definieron varios cálculos de objetos con tipos. En los primeros trabajos ([FHM94], [FM95a], [FM95b]) la idea fue representar la especialización de métodos en la herencia. En los más recientes se busca proveer las características más útiles del modelo basado en clases a partir de cálculos con construcciones más primitivas que las clases.

En [BF98] y [FM98] se definen cálculos imperativos de objetos y funciones. Extienden el lambda-cálculo con expresiones relacionadas a objetos (primitivas de objetos para enviar mensajes, modificar y extender objetos, primitiva let para expresar secuencia) y expresiones relacionadas a tipos (cuantificadores de tipos y una forma sintáctica para expresar tipos abstractos de datos). En [BF98] se define una semántica imperativa similar a la de Abadi y Cardelli, utilizando un store global y entornos locales para mantener las referencias a otros objetos. Una diferencia con el impς-cálculo es que como se extiende el cálculo lambda, existen valores o resultados de evaluación que no son objetos (sino por ejemplo lambda abstracciones) y esto determina una variedad en las cosas a almacenar en el store. El sistema de tipos definido por Bono y Fisher distingue dos especies de objetos: los prototipos (tipados con expresiones de tipos **pro-.**) y los objetos (tipados con expresiones de tipos **obj-**). Los prototipos pueden extenderse, modificarse y recibir mensajes, pero sólo soportan subtipado elemental. Los objetos no pueden ser extendidos ni modificados, pero sí tienen subtipado más complejo (a lo ancho y en profundidad). Esto es así por los problemas de inconsistencia de tipos que surgen al combinar extensión de objetos con subsunción. Los prototipos pueden mutar a objetos, "sellándose", es decir que en adelante no pueden ser extendidos.

En los cálculos definidos en estos trabajos hay un énfasis fuerte puesto en los sistemas de tipos, debido a que gran parte de la expresividad que se busca representar es implementada usando construcciones primitivas de tipos. Esta es una gran diferencia con nuestro trabajo que, basado en un cálculo no tipado, busca definir un marco para representar todas las características propias del paradigma sólo a partir de objetos y mensajes. Por ejemplo en [BF98], parte del encapsulamiento necesario para representar clases es provista con un cuantificador existencial sobre tipos (rows) que permite ocultar parte de la implementación de un objeto. También se utiliza un cuantificador universal de tipos (rows) para expresar polimorfismo en los métodos; es necesario que los métodos de un objeto sean polimórficos sobre el tipo de *self* para poder aplicarse luego de que el objeto haya sido extendido con métodos nuevos.

Otra característica distintiva es que representan el comportamiento compartido a partir de la extensión funcional de los objetos.

La definición de subclases se hace extendiendo la superclase con nuevos mensajes. En el cálculo lambda de objetos de Fisher-Honsell-Mitchell, este tipo de extensión se deriva de la semántica funcional del cálculo. En el cálculo imperativo de [BF98] el operador de extensión definido para los métodos actúa sobre un clon del objeto a extender. Esto es necesario para conservar la consistencia de tipos de los objetos, pero tiene la desventaja de que mantiene al

objeto original intacto, con lo que los otros objetos que lo referencian no se enteran de la extensión que ocurrió.

A diferencia de los cálculos de esta línea, con el cálculo definido en nuestro trabajo se puede representar herencia, delegación y otros esquemas de sharing a partir de una semántica puramente imperativa, y sin construcciones primitivas especiales para sharing. El operador de extensión que definimos es puramente imperativo, en el sentido de que permite agregar mensajes a un objeto de manera que los objetos que lo referenciaban previamente puedan enviarle los nuevos mensajes agregados. Esto es muy útil para la modelización de lenguajes de programación reales, que tienen construcciones de este tipo.

8.2 ς-cálculo funcional con extensiones

Luigi Liquori extendió uno de los cálculos funcionales de Abadi y Cardelli para permitir extensión de objetos. La adición de nuevos mensajes se realiza mediante el operador $\Leftarrow$. La semántica del operador $\Leftarrow$ en la extensión es la misma que en nuestro cálculo (agrega el mensaje si no pertenece al protocolo del objeto, y actualiza el método si pertenece), con la diferencia de que este cálculo es funcional.

El cálculo base es el $Obj_{1<:}$, que es un cálculo tipado de primer orden. La extensión se llama $Obj1_{<:}^{+}$. La sintaxis del nuevo cálculo es igual a la del cálculo original, se extiende la semántica y el sistema de tipos. La extensión definida es conservativa. Para poder tratar adecuadamente los tipos de la extensión de objetos, el cálculo distingue dos especies de tipos de objetos, que distinguen los objetos que pueden ser extendidos de los que no.

En este trabajo también se presenta una codificación de clases como objetos, representando clases, metaclases e instancias en forma similar a Smalltalk-80. Esta codificación es mejor que la que se realiza en el cálculo sin extensión (descripta en el capítulo 5), ya que permite crear las instancias de una subclase realizando actualizaciones y extensiones de una instancia creada por su superclase. Esta forma de creación es similar a la creación y especialización de los objetos de Asterix, que se presentó en la sección 7.2. Sin embargo, una subclase no se puede crear actualizando una clase, ya que el sistema de tipos no lo permite, y lo mismo sucede con las metaclases. La codificación de clases presentada en este trabajo también se puede realizar en nuestro cálculo, y al ser no tipado, permitirá crear una subclase actualizando y extendiendo su superclase.

Cabe destacar que nuestro trabajo fue independiente del de Liquori.

8.3 Cálculo para modificación encapsulada de objetos

Otro trabajo muy relacionado con el nuestro es [MVM96], desarrollado por Mens, De Volder y Mens.

En un entorno de prototipos, cuando un objeto incorpora parte del comportamiento de otro, necesita conocer como éste otro implementa sus mensajes. Si un objeto que es cliente de otro tiene las mismas posibilidades de acceder a su implementación que uno que mantiene una relación de sharing, entonces el encapsulamiento será violado. El objetivo de su trabajo es dar un modelo formal de la delegación que solucione los problemas que se presentan en relación con el encapsulamiento. Para esto, definen un cálculo en donde hay una forma de sharing primitiva.

Dado que el cálculo que definen es funcional, el comportamiento del donor se incorpora en el objeto borrower, y las futuras modificaciones del donor no afectan al borrower. Por esta razón, el efecto de esta forma de sharing es más parecido a lo que en nuestra clasificación llamamos embedding implícito que a la delegación. Sin embargo, en esta sección usaremos la palabra delegación en el sentido de tomar comportamiento por defecto de otro objeto, como esta usada en estos trabajos.

En el resto de la sección describiremos este trabajo y presentaremos una forma de traducir su cálculo en el impEς-cálculo.

En [MVM96] se define el "principio inmaculado de la interface a clientes", que dice que se debe poder distinguir entre los clientes comunes (los que le envían mensajes) y los clientes de especialización (los que reúsan el comportamiento), y que cada uno tiene que poder acceder a una interface diferente. Enfatiza que los clientes comunes no deben poder acceder a la interface de especialización.

Para respetar este principio, definen dos tipos diferentes de modificación incremental: herencia encapsulada (con late binding del self), que hace uso de la interface de especialización, y modificación conservativa (sin late binding del self), que hace uso de la interface de cliente. Los clientes sólo pueden hacer uso de la modificación conservativa, con lo cual no se viola el principio de la interface a clientes.

En su modelo hay dos tipos de entidades: los generadores y los objetos. Los objetos son conjuntos de métodos, los generadores son moldes para objetos que definen el comportamiento de éstos.

El uso de los generadores esta muy restringido: no pueden ser pasados como parámetros, ni recibir mensajes; sólo sirven para definir a los objetos y sus relaciones de sharing. Las referencias a self en los generadores no están ligadas a ningún objeto. Manipulando las referencias a self en los generadores se logra tener modificaciones que no respeten en encapsulamiento previsto para los clientes, con el objeto de obtener la herencia encapsulada.

La sintaxis del Δ-cálculo tiene dos capas: una define objetos y pasaje de mensajes, y la otra expresa la manipulación explícita de generadores. Se define de la siguiente manera:

Gen ::= ε | Ident(Ident) = Ident#Obj | Ident | Gen ; Gen |>Obj<

Obj ::= Obj.Ident(Obj) | [Gen] | Ident | [Gen Δ Gen]

ε representa el generador vacío.

Ident(Ident) = Ident#Obj representa un generador con un único mensaje. El binder **#** tiene la misma semántica que el ς de Abadi y Cardelli. El pasaje de parámetros en este cálculo es primitivo.

Un generador ***Ident*** es una referencia a self.

Un generador puede ser compuesto con otro mediante **;**. ***Gen1;Gen2*** corresponde a la concatenación con preferencia a derecha de los métodos de ambos generadores. Este es el operador que se utiliza para hacer modificaciones, ya que si *Gen1* y *Gen2* tienen mensajes en común *Gen1;Gen2* responde a estos mensajes con los métodos definidos por *Gen2*.

>Obj< devuelve el generador correspondiente al objeto ***Obj***. El self en ***>Obj<*** se fija y cuando ***>Obj<*** se extiende con **;** el objeto referenciado por self no es modificado.

Sólo los objetos pueden recibir mensajes, y ser pasados como parámetros. La semántica del pasaje de mensajes ***Obj1.Ident(Obj2)*** es buscar y evaluar el mensaje Ident en el objeto receptor *Obj1*, pasando el objeto *Obj2* como parámetro.

Ident es una referencia a un argumento dentro de un método cuyo argumento tiene ese nombre. Se evalúa buscando ese identificador en el entorno en el que aparece referenciado.

[Gen] representa la creación de un objeto a partir del generador *Gen*.

El operador Δ construye un objeto a partir de dos generadores. El término *Gen1* Δ *Gen2* define un objeto en el cual los mensajes enviados a él se buscarán en *Gen1* y las referencias internas a self se buscarán en *Gen2*. Así, *Gen1* se comporta como si fuera una interface a clientes y *Gen2* como una interface de especialización.

El operador Δ representa la ligadura explícita del self en un generador, y sirve para definir delegación.

[Gen] es una abreviatura para *[Gen* Δ *Gen]*.

Se define una semántica denotacional para estos operadores. La noción de identidad no existe en este cálculo.

Modificación Conservativa

La modificación conservativa se realiza usando el operador >< para tomar el generador de un objeto y modificando éste generador con ;. En *>o<* las referencias a self se resolverán en el objeto original *o*, ignorando posteriores modificaciones.

Veamos un ejemplo tomado de [MDM96],

```
[       cartesiano(arglist) =
      [ getx = arglist.x;
                gety = arglist.y;
                distancia = self # [self].sumaDeCuadrados.sqrt;
                sumaDeCuadrados = self # [self].getx.sqr.add([self].gety.sqr)   ];
      puntoCartesiano = Env # [Env].cartesiano [x = 3; y = 4]
]
```

Este objeto tiene dos mensajes: *cartesiano* recibe las coordenadas de un punto en el plano y devuelve un punto que puede calcular su distancia cartesiana; y *puntoCartesiano* devuelve el punto de coordenadas (3,4) que sabe calcular su distancia cartesiana.

Supongamos que queremos modificar la respuesta al mensaje distancia para que compute la distancia de Manhattan en el punto de coordenadas (3,4). Agregamos un mensaje *puntoManhattan*, que toma el punto cartesiano (3,4) y sobreescribe el método *distancia*.

```
[cartesiano(arglist) =
      [ getx = arglist.x;
        gety = arglist.y;
        distancia = self # [self].sumaDeCuadrados.sqrt;
        sumaDeCuadrados = self # [self].getx.sqr.add([self].gety.sqr)   ];
puntoCartesiano = Env # [Env].cartesiano [x = 3; y = 4];
puntoManhattan = Env #
        [       >[Env]. puntoCartesiano<;
                distancia = Self # [Self].getx.abs.add([Self.gety.abs])   ]
]
```

Esta modificación es conservativa porque incorpora el objeto *[Env].puntoCartesiano* tal como esta definido, no lo cambia ni necesita conocer su estado interno.

Herencia Encapsulada

La herencia encapsulada es otro mecanismo de modificación, que consiste en agregar o modificar métodos al generador que da origen a un objeto desde sus propios métodos, referenciándolo como self. Este tipo de modificaciones sólo pueden hacerse en los métodos del objeto a modificar, ya que el self sólo es alcanzable desde los métodos propios del objeto.

Consideremos, por ejemplo, a un objeto que representa una persona:

```
[       nombre = "Angela";
        sexo = "femenino";
        título = "Srta";
        EncabezadoDeCarta = Self # [Self]. título.concatenar([self].nombre)
]
```

Y supongamos que queremos extender el comportamiento de este objeto de forma que pueda ser usado como molde de nuevos objetos similares. Para esto, vamos a definir un nuevo método *nuevaPersona*, que recibe el nombre y sexo de la nueva persona.

```
[       nombre = "Angela";
        sexo = "femenino";
        título = "Srta";
        encabezadoDeCarta = self # [self]. título.concatenar([self].nombre)
        nuevaPersona(init) = self #
                [self; nombre = init.nombre ; sexo = init.sexo;
                título = sexo.equal ("femenino").if ([then = "Srta"; else = "Srto"])
                ]
]
```

El método nuevaPersona *de este ejemplo no podría haberse implementado usando modificación conservativa (con ><). Si se usara modificación conservativa, al ejecutar el método* encabezadoDeCarta *en una persona creada con* nuevaPersona, nombre *se evaluaría sin tener en cuenta la modificación: todas las personas tendrían* Angela *en su encabezado.*

Consideremos otra implementación de Persona, que calcule el título dependiendo del sexo. En este caso la implementación de *nuevaPersona* cambia.

```
[       nombre = "Angela";
        sexo = "femenino";
        título = Self # [Self].sexo.equal("femenino").if ([then="Srta."; else="Srto."])
        encabezadoDeCarta = Self # [Self]. título.add([self].nombre)
        nuevaPersona(init) = Self # [ Self; nombre = init.nombre; sexo = init.sexo]
]
```

En este otro ejemplo, si el método *NuevaPersona(init)* estuviera implementado usando modificación conservativa, en una persona creada con ese método los llamados dentro del método *título* se resolverían sin considerar el nuevo *nombre* y *sexo*. Esta es la razón por la que necesitamos usar herencia encapsulada.

En estos dos ejemplos vimos que al cambiar la implementación del objeto original que representa una persona, la implementación del método *nuevaPersona* tuvo que cambiar también. Esto pone en evidencia que la implementación de *nuevaPersona* necesita conocer detalles de implementación del objeto persona. En forma consecuente con esto, la modificación

conservativa no es suficiente para implementar *nuevaPersona*, se necesita otro mecanismo de modificación: herencia encapsulada.

Representación de los operadores del Δ-cálculo en el ς-cálculo

Todas las construcciones del Δ-cálculo se pueden implementar usando como base el impEς-cálculo. Como el impEς-cálculo sólo tiene objetos y mensajes, los generadores serán imitados por objetos.

Cada objeto tendrá un mensaje para cada operador en el cálculo (; , >< y Δ). Para hacer estas construcciones, supongamos que cada objeto en el cálculo esta implementado en el estilo especificación-implementación, o sea con premétodos, como en los capítulos anteriores.

Operador ; de composición

El operador ; permite agregar o modificar los métodos de un objeto. Para implementar este operador agregamos a cada objeto *o1* un mensaje que recibe otro objeto *o2* y lo extiende con los métodos del objeto receptor *o1*.

El objeto $o = [m_i{:}\varsigma(x_i)b_i{}^{\,i \in 1..n}]$ extendido con el operador ; se expresa en el cálculo así:

$$
\begin{aligned}
o' = [\ & m_i{:}\varsigma(x_i)b_i{}^{\,i \in 1..n}, \\
& ;' = \varsigma(x)\,\lambda(o) \quad \text{let } m_1code = x.{*}m_1 \text{ in} \\
& \qquad\qquad \text{let } m_2code = x.{*}m_2 \text{ in} \\
& \qquad\qquad \ldots \\
& \qquad\qquad (\ldots(o.m_1 \Leftarrow \varsigma(z)z.{*}m_1(z)).{*}m_1 \Leftarrow \varsigma(z)m_1code) \\
& \qquad\qquad\qquad\qquad \Leftarrow \varsigma(z)z.{*}m_2(z)).{*}m_2 \Leftarrow \varsigma(z)m_2code)\ldots) \\
&]
\end{aligned}
$$

*o1;o2*se traduce en el cálculo como *o2.;'(clone(o1))*.

Una sintaxis mas parecida se puede obtener instalando otro método en cada objeto que invierta el orden de los argumentos, mensaje $; = \varsigma(z)\lambda(o)o.;'(z)$. Con este mensaje *o1;o2* se puede traducir como (clone(*o1*)).;(*o2*).

Operador >< para obtener el generador de un objeto

La característica de este operador es que cuando un generador obtenido a partir de un objeto mediante >< es extendido, las referencias a self siguen referenciando al generador original (no al extendido, no hay late binding del self). Esto se puede simular construyendo para el objeto

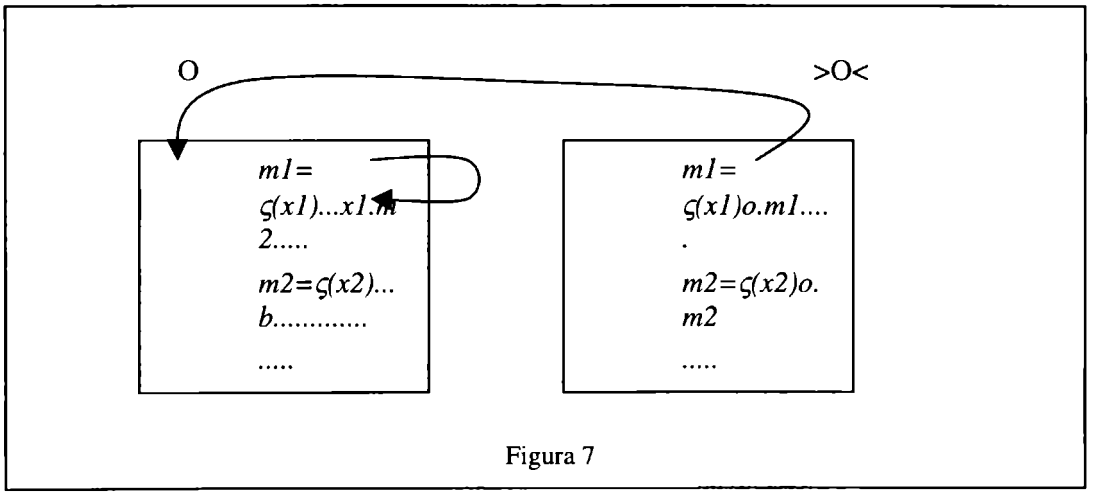


Figura 7

una cáscara con el mismo protocolo que el objeto original, donde cada mensaje es reenviado al objeto original. Dado un objeto o, $>o<$ tendrá la forma mostrada en la Figura 7.

Así, cuando se agrega un mensaje al objeto $>o<$, desde su método se puede acceder a todo el protocolo, y cuando se modifica un mensaje existente también a un mensaje también, pero las viejas referencias en los viejos métodos sólo pueden ver el comportamiento original.

El objeto $o= [m_i:\varsigma(x_i)b_i{}^{i \in 1..n}]$ extendido con el operador $><$ será:

$$o' = [m_i:\varsigma(x_i)b_i{}^{i \in 1..n},\ >< = \varsigma(x)[\ m_i:\varsigma(x_i)x.m_i{}^{i \in 1..n}]\ ]$$

$>o<$ se traduce en el cálculo como $o.><$.

Operador Δ de delegación

El operador de delegación liga explícitamente el self de un generador.

A cada objeto se agrega un mensaje que devuelve el objeto a ser usado como self. Como los objetos están implementados con especificación e implementación, el objeto *self* es usado como parámetro cuando se invoca la implementación El método Δ modifica el campo *self* con un nuevo valor.

El objeto $o= [m_i:\varsigma(x_i)b_i{}^{i \in 1..n}]$ extendido con el operador Δ será:

$$[*m_i=\varsigma(x)\ \lambda(rec)\ b_i\ \{\ x_i \leftarrow rec\},\ m_i=\varsigma(x)\ x.*m_i(x.self),\ self = \varsigma(x)\ x,\ \Delta= \varsigma(x)\ \lambda(es)\ x.self \Leftarrow es\ ]$$

$[g1\ \Delta\ g2]$ se traduce en el cálculo como $clone(g1).\Delta(g2)$.

Función de traducción

Entre los objetos del impEς-cálculo distinguiremos dos tipos: los que representan a los objetos del Δ-cálculo y los que representan los generadores del Δ-cálculo.

Los generadores pueden responder a los mensajes Δ y ;. Los objetos pueden responder al mensaje $><$.

Si *term* es un término del Δ-cálculo, *{ term }* es la representación de *term* en el impEς-cálculo.

```
{m(x) = z#o} = [*m = ς( _ ) λ(z) λ(x) {o},
                Δ = ς(x) λ(g2) [m = ς(y) x.*m (g2)
                                  >< = ς(z)(clone(x)) . Δ ⇐ λ(g2) z ]
                ;' = ς(y) λ(g) let mcode = y.*m in
                        let old;' = g.;' in
                        let oldΔ = g. Δ in
                                g.*m ⇐ ς( _ ) mcode
                                . Δ ⇐ ς(x) λ(g2) oldΔ (g2).m ⇐ ς( _ ) x.*m(g2)
                                .>< = ς(z) x. Δ ⇐ λ(g2) z
                                . ;'⇐ ς( _ ) λ(g2) old;'(g2) .*m ⇐ ς( _ ) mcode
                                        . Δ ⇐ ς(x) λ(g2) oldΔ (g2).m ⇐ ς( _ ) x.*m(g2)
                                .>< = ς(z) x. Δ ⇐ λ(g2) z
                ; = ς(y) λ(z) z .;'(y)
                ]

{ g1 ; g2 } = {clone(g1)} .; ({g2})

{>o<} = {o}. ><

{[g1 Δ g2]} = {g1}.Δ ({clone(g2)})
```

Capítulo 9

Hacia una Relación de Comportamiento

9. Hacia una Relación de Comportamiento

En el desarrollo de esta tesis, nos encontramos con el problema de querer demostrar que dos objetos se comportan de la misma manera, en el sentido de que uno puede reemplazarse por el otro. Este problema no se pudo resolver, pero se investigaron algunas líneas de solución, cuyo seguimiento queda como trabajo futuro. En esta sección explicaremos el problema y los intentos para solucionarlo.

Una forma de probar que la representación de sharing que proponemos es correcta, es comparar un objeto de la representación con otro más simple que tenga el comportamiento que se espera del de la representación, y demostrar que se comportan de la misma manera. Por ejemplo:

- Un objeto independiente escrito en estilo especificación - implementación (con premétodos) se comporta igual que un objeto definido directamente (sin premétodos).
- Un objeto que incorpora un método de otro se comporta igual que si definiera ese método como propio.
- Un objeto que delega un mensaje en otro se comporta como si definiera el método, siempre que el donor no se modifique
- Si se modifica un método de un objeto donor, un objeto que delega en él se comportará como si incoporara el nuevo método.
- Si se extiende un donor, un objeto que delega en él se comporta como si hubiese sido extendido de la misma manera.

En estos casos, lo que se quiere es que el primer objeto (el que se quiere ver que es correcto) pueda utilizarse en lugar del segundo, pero no se requiere que esto suceda a la inversa. Decimos que el primer objeto puede *simular* al segundo. En general, los objetos de la representación tendrán más habilidades que los comunes, ya que a diferencia de estos, pueden responder a los mensajes de implementación.

Dado que definimos el comportamiento de un objeto como el conjunto de mensajes a los que puede responder y la respuesta a esos mensajes, la primer idea de que un objeto pueda simular a otro sería que pueda responder al menos a todos los mensajes que el otro puede responder, y además que el objeto que se obtenga como respuesta de enviarle el mensaje al primero pueda simular al que se obtiene como respuesta del segundo.

Comparación de términos

Con esta idea intentamos definir la relación R de *puede simular a* sobre los términos del cálculo. Como la idea de la relación es que los objetos relacionados responden de manera similar a los mismos mensajes, los casos que tienen una definición especial son los términos que construyen objetos, (con el constructor $[\ ]$), y los que envían mensajes (términos de la forma $o.m$). Los demás términos se comparan envíandoles todos los mensajes posibles, y verificando que sus respuestas estén relacionadas. La definición obtenida no es inductiva, ya que en este último caso, para comparar un par de términos, se deben comparar términos más grandes.

Veamos la comparación entre dos términos que definen objetos con el constructor $[...]$. Para que

$$[l_i=\varsigma(x_i)b_i^{\,i\in M1}]\ R\ [l_j=\varsigma(z_j)c_j^{\,j\in M2}]$$

debería cumplirse que M_2 esté incluido en M_1 (es decir, que el primer objeto entiende al menos todos los mensajes del segundo), y que para todo $k \in M_2$, $b_j \; R \; c_j$ (las respuestas correspondientes a todos los mensajes que entiende el segundo objeto están relacionadas). Según esta definición, el objeto $[\ ]$ que no entiende ningún mensaje, puede ser simulado por cualquier otro objeto.

Para comparar un término que es un pasaje de mensaje con algún otro término, $o_1.m \; R \; o_2$, lo que se quiere ver es que la respuesta al mensaje pueda simular al otro término. Para saber cuál es esa respuesta, es necesario evaluar o_1, y tomar el cuerpo del método para el mensaje m. Para poder realizar esta evaluación, se debe contar con el contexto de evaluación, es decir, con I, σ y S correspondientes. Entonces, definimos la relación sobre tuplas $(o\ ,\ I,\ \sigma,\ S\ )$. La tupla $(o_1,\ I_1,\ \sigma_1,\ S_1)$ estará relacionada con $(o_2\ ,\ I_2,\ \sigma_2,\ S_2)$ si

$I_1, \sigma_1, S_1 |\!- o_1 \sim\!> id.I_1', \sigma_1'$

$\sigma_1'(m(I_1'(v_1))) = \varsigma(x)b$ (el cuerpo del mensaje m en el resultado de evaluar o_1 es b y la variable ligada es x),

y

$(b,\ I_1,\ \sigma_1',\ S_1, x \rightarrow v_1) \; R \; (o_2,\ I_2,\ \sigma_2,\ S_2)$.

De manera simétrica se define la relación para $(o_1, I_1, \sigma_1, S_1) \; R \; (o_2.m, I_2, \sigma_2, S_2)$.

Para los demás términos, la relación se define como $(o_1, I_1, \sigma_1, S_1) \; R \; (o_2, I_2, \sigma_2, S_2)$ si para todo mensaje m,

$(o_1.m, I_1, \sigma_1, S_1) \; R \; (o_2.m, I_2, \sigma_2, S_2)$

Otro caso especial a considerar es el de las variables. Hasta aquí las variables quedarían contempladas en el último caso que describimos. Ahora veamos el siguiente caso: queremos ver si $[m=\varsigma(x)x]$ puede simular a $[m=\varsigma(y)y]$. Intuitivamente vemos que sí, ya que ambos objetos tienen un único mensaje que los retorna a ellos mismos. Con la definición de la relación vista hasta ahora, estos términos no se pueden relacionar. Para solucionar estos casos, queremos relacionar en el caso de los términos que definen objetos a las variables ligadas a ς, que representan al mismo objeto. Para esto agregamos a la relación un conjunto A de pares de variables. Los pares que pertenecen a este conjunto se considerar relacionados por R. Entonces, en el caso de las variable tenemos que $(\ x,\ I_1,\ \sigma_1\ ,\ S_1)\ \mathrm{R_A}\ (y,\ I_2,\ \sigma_2\ ,\ S_2)$ *si*

$(x,y) \in A$,

ó

$(x,y) \notin A$ y $\forall m \;\; (x.m, I_1, \sigma_1, S_1)\ \mathrm{R_A}\ (y.m,\ I_2\ ,\ \sigma_2\ ,\ S_2)$.

La definición para los términos constructores, teniendo en cuenta el conjunto A y los contextos de evaluación, es la siguiente: $(\ [l_i=\varsigma(x_i)b_i\ ^{i \in M1}]\ ,\ I_1,\ \sigma_1,\ S_1)\ R_A\ (\ [l_i=\varsigma(z_j)c_j\ ^{j \in M2}]\ ,\ I_2,\ \sigma_2,\ S_2)$ si

$M_1 \subseteq M_2$

$I_1\ ,\ \sigma_1,\ S_1 \mid\!- [l_i=\varsigma(x_i)b_i\ ^{i \in M1}] \sim\!> id_1\ .I_1', \sigma_1'$

$I_2\ ,\ \sigma_2,\ S_2 \mid\!- [l_i=\varsigma(x_i)b_i\ ^{i \in M2}] \sim\!> id_2.I_2'.\sigma_2'$

$(x_i\ ,z_j) \in A,\ \forall i \in M_1,\ \forall j \in M_2$

$\forall$

$k \in M_1 : (b_k,\ I_1'\ ,\ \sigma_1'\ ,\ (S_1, x_k \rightarrow id_1))\ R_A\ (c_k,\ I_2'\ ,\ \sigma_2'\ ,\ (S_2, z_k \rightarrow id_2))$

La condición de que $(x_i,\ z_j) \in A$, es la que nos permitirá considerar que se refieren a objetos relacionados dentro de la prueba de que los métodos de los dos objetos están relacionados.

Esta definición es muy compleja y resulta poco clara. Además, como la definición no es inductiva, no es claro que la relación quede precisamente definida. A continuación veremos otra forma de definirla, más clara y concisa.

Simulación y bisimulación para objetos

Otro intento de representar la relación de comportamiento entre objetos fue a partir de elementos del dominio semántico del cálculo. Tomamos la técnica de simulación y bisimulación, usada en semántica de sistemas concurrentes. La simulación se usa para expresar que el comportamiento exhibido por un proceso también puede ser exhibido por otro. Bisimulación se usa para decir que dos procesos exhiben el mismo comportamiento. Nosotros intentaremos usar simulación para expresar cuando un objeto exhibe al menos el mismo comportamiento que otro, y bisimulación para expresar cuando un objeto exhibe exactamente el mismo comportamiento que otro. [Gordon96] define una forma de bisimulación para objetos en uno de los cálculos funcionales de [AC96]. El uso de estas técnicas provee una definición más clara y concisa para la relación de comportamiento entre objetos que estamos buscando.

Nosotros intentamos definir simulación y bisimulación para objetos en el cálculo imperativo impς-cálculo de Abadi y Cardelli [ABPPR98].

La definición general de simulación dice que una simulación es una relación **R** sobre un cierto dominio (como programas, estados o términos) que satisface la siguiente condición: si dos elementos P y Q están relacionados, entonces para cada acción observable a que P pueda realizar, transformándose en P', Q puede realizar la misma acción a transformándose en Q', y P'está relacionado con Q'. La Figura 8 ilustra la situación

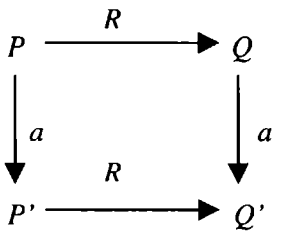


Figura 8

Se dice que P puede ser simulado por Q si existe una simulación que relaciona P y Q.

Para aplicar esta definición a la relación que nosotros buscamos, debemos establecer cuáles son los elementos del dominio de la relación, y cuáles sus acciones observables. En principio tomaremos a los elementos de la relación como objetos.

Las acciones que primero aparecen sobre objetos es el pasaje de mensajes, por lo tanto ese será el primer tipo de acciones observables, que llamaremos "message-send". El esquema de la definición de simulación se lee entonces: si un objeto P puede ser simulado por otro objeto Q, entonces para todo mensaje m que P pueda entender retornando como resultado un objeto P', Q también puede entender el mensaje m, en respuesta al cual retorna el objeto Q', y P' puede ser simulado por Q'.

Como el formalismo en que estamos trabajando tiene una semántica imperativa, los objetos pueden cambiar como resultado de responder a mensajes. Queremos reflejar esta característica en nuestra definición de simulación sobre objetos. Pretendemos que si un objeto puede simular a

otro y ambos reciben un mensaje, luego de haber respondido al mensaje, se mantenga la relación de simulación entre ellos. Esto determina el segundo tipo de acciones observables sobre objetos, que llamamos "after". Si un objeto P puede ser simulado por otro objeto Q, entonces para todo mensaje m que P entiende, si luego de responder a m se transforma en P', Q también puede entender el mensaje m, luego de responderlo se transforma en Q', y P' puede ser simulado por Q'.

Queremos que la relación de igualdad de comportamiento que estamos buscando permita sustituir un objeto por otro, dentro de un entorno. Para que la simulación que estamos definiendo cumpla ese requisito, debemos asegurar que si un objeto O_1 puede ser simulado por otro O_2, entonces el resultado de usar O_1 en un entorno pueda ser simulado por el resultado de usar O_2 en ese entorno. En nuestro cálculo, un entorno para un objeto es el término (o programa) en que está definido. El objeto y su entorno interactúan a través de las operaciones básicas del cálculo. Por ejemplo el objeto puede estar contenido en un término que le envía un mensaje. Otro tipo de interacción es la actualización; como el impς-cálculo no tiene encapsulamiento, un objeto puede ser modificado desde el exterior. Por lo tanto debemos considerar las actualizaciones dentro de las acciones observables entre un objeto y su entorno. Si no consideramos este tipo de operaciones, entonces dos objetos relacionados podrían comportarse diferente dentro de un entorno que los actualiza.

En consecuencia, el tercer tipo de acciones observables para la definición de simulación sobre objetos son los "update". Si un objeto P puede ser simulado por otro objeto Q, entonces para todo mensaje m de P, si al cambiar el método asociado a m, P se transforma en P', al actualizar Q de la misma manera se obtiene Q', y P' puede ser simulado por Q'.

Es importante destacar que la inclusión del "update" en la definición de simulación introduce complicaciones. Si bien la modificación de objetos desde el exterior no es una característica propia del paradigma, es necesario contemplarla dentro de las acciones a observar porque proviene del formalismo con que trabajamos.

Para precisar la definición en términos formales, empezaremos por especificar cómo son los elementos del dominio de la relación. Usamos pares de elementos del dominio semántico del impς-cálculo. La evaluación de un término en el cálculo da un resultado y un store. El dominio de la relación de simulación es por lo tanto el conjunto de pares

$$(r, \sigma)$$

donde r es un resultado (un objeto evaluado), y σ el store correspondiente. También se puede ver a estos pares como un mundo de objetos y un objeto distinguido dentro de él. La relación especificará que un cierto objeto dentro de un mundo puede ser simulado por un cierto objeto en otro mundo.

El esquema de la simulación para objetos se muestra en la Figura 9, donde la acción a es de cualquiera de los tres tipos de acciones explicados.

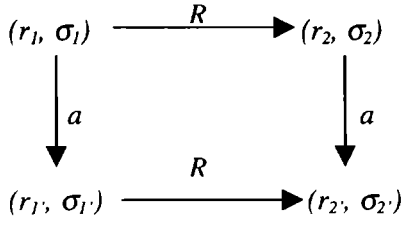


Figura 9

Para especificar los tipos de acciones descriptos anteriormente usamos términos del impς-cálculo llamados contextos, como en [GR96]. Un contexto es un término con una sola variable libre distinguida que se usa para denotar un "agujero" en un programa (notada con "-") que representa al objeto. Tendremos por lo tanto tres tipos de contextos, correspondientes a los tres tipos de acciones:

Contexto	*Acción observable*
$-.m$	Pasaje de mensajes ("message-passing")
Let $-.m$ *in* -	Observación del receptor luego de responder a un mensaje ("after")
$-.m \Leftarrow \varsigma(x)b$	Actualización del objeto ("update")

Los pares (r, σ) se relacionan a través de estos tres tipos de acciones. Decimos que un par (r, σ) está relacionado con otro par (r', σ') por una acción a, si (r', σ') se obtiene de evaluar el contexto correspondiente a a en el store σ, ligando la variable libre - al resultado r.

Esto se puede formalizar en un sistema de transición etiquetado, dado por el conjunto de reglas:

$$\frac{\sigma, - \rightarrow r1 \mid\!- E \sim> r_2 . \sigma'}{(r_1, \sigma) \rightarrow^{E} (r_2, \sigma')}$$

donde E es un contexto de alguna de las tres formas definidas.

En este marco, formulamos la definición de una simulación para objetos:

Definición

- Una *simulación* es una relación **R** sobre *resultados* x *stores* que satisface:

  Si (a, σ_1) **R** (b, σ_2) entonces para toda acción E tal que $(a, \sigma_1) \rightarrow^{E} (c, \sigma_1')$, existe (d, σ_2') tal que $(b, \sigma_2) \rightarrow^{E} (d, \sigma_2')$ y (c, σ_1') **R** (d, σ_2').

- Decimos que (a, σ_1) puede ser simulada por (b, σ_2) si existe una simulación **R** que relaciona (a, σ_1) con (b, σ_2).

En el mismo marco, definimos bisimulación:

Definición

- Una *bisimulación* es una relación **R** sobre *resultados* x *stores* que satisface:

  Si (a, σ_1) **R** (b, σ_2) entonces

  1. para toda acción E tal que $(a, \sigma_1) \rightarrow^{E} (c, \sigma_1')$, existe (d, σ_2') tal que $(b, \sigma_2) \rightarrow^{E} (d, \sigma_2')$ y (c, σ_1') **R** (d, σ_2').
  2. Para toda acción E tal que $(b, \sigma_2) \rightarrow^{E} (d, \sigma_2')$, existe (c, σ_1') tal que $(a, \sigma_1) \rightarrow^{E} (c, \sigma_1')$ y (c, σ_1') **R** (d, σ_2').

- Decimos que (a, σ_1) es bisimilar a (b, σ_2) si existe una bisimulación **R** que relaciona (a, σ_1) con (b, σ_2).

Como explicamos anteriormente, queremos que la relación definida permita sustituir objetos relacionados en un entorno, manteniendo el comportamiento general del sistema. Podemos especificar este requerimiento en una propiedad más fuerte, que dice que si un término con variables libres es evaluado en dos stores diferentes, y cada variable libre está ligada en el primer store a un objeto que puede ser simulado por el objeto a que está ligada en el segundo store, entonces el resultado de la evaluación del término en el primer store puede ser simulado por el resultado de la evaluación en el segundo store.

Propiedad

Sea b un término del cálculo, S_1 y S_2 pilas, σ_1 y σ_2 stores. Si para cada variable x libre en b, se cumple que

$S_1(x) = v_1$

$S_2(x) = v_2$

(v_1, σ_1) puede ser simulado por (v_2, σ_2)

entonces si

$\sigma_1, S_1 \vdash b \sim> r_1 . \sigma_1'$ y

$\sigma_2, S_2 \vdash b \sim> r_2 . \sigma_2'$,

(r_1, σ_1') puede ser simulado por (r_2, σ_2')

Aún no demostramos la validez de esta propiedad. Creemos que una forma de hacerlo puede ser por inducción sobre la estructura de b, probando que vale cuando b es una variable, una definición de objeto, un envío de mensajes, una actualización o una construcción let.

La validez de esta propiedad asegurará que objetos relacionados a través de las relaciones definidas pueden sustituirse en un programa sin alterar la semántica global. Una noción de igualdad de comportamiento como la que se intenta definir es útil, no sólo para probar que las representaciones que presentamos en este trabajo son correctas, sino también en otras aplicaciones.

Muchas veces durante el ciclo de desarrollo de software es importante determinar si un objeto puede ser reemplazado por otro. En una fase exploratoria, por ejemplo, al prototipar un sistema o un módulo se definen objetos simples con el mínimo comportamiento requerido. En una etapa posterior esos objetos simples se reemplazan por otros cuya implementación está mas refinada y completan la funcionalidad con comportamiento adicional. Los nuevos objetos deben emular la funcionalidad ya implementada por el prototipo, y a la vez proveer la implementación para el resto.

También, durante la evolución de un sistema, para mejorar la performance de un módulo, surge la necesidad de cambiar algunos objetos por otros con una implementación más eficiente. En este caso es necesario asegurar que los nuevos objetos tengan exactamente el mismo comportamiento que los reemplazados, para que no se altere la funcionalidad del sistema total.

En los dos ejemplos anteriores se requiere contar con una relación entre objetos que sirva para determinar cuándo uno puede ser reemplazado por otro. Se podría decir que un objeto puede reemplazar a otro si exhibe "al menos el mismo comportamiento", como en el ejemplo de la prototipación. También se podría fortalecer la condición estableciendo que un objeto puede sustituir a otro si exhibe "exactamente el mismo comportamiento", en cuyo caso cualquiera de

los dos se puede reemplazar por el otro, como en el segundo ejemplo. La noción de simulación entre objetos puede usarse para caracterizar idea de exhibir "al menos el mismo comportamiento". La bisimulación captura la idea de exhibir "exactamente el mismo comportamiento".

La investigación de relaciones de igualdad de comportamiento entre objetos es una de las líneas de continuidad principales de este trabajo. Pensamos que es un problema interesante con una aplicación muy amplia.

Capítulo 10

Conclusiones

10. Conclusiones

El objetivo de esta tesis fue encontrar una formalización de aspectos relevantes del paradigma de Orientación a Objetos que se base en un conjunto minimal de conceptos y en la que las otras características del paradigma estén construidas sobre estos elementos básicos.

Para ello, primero se hizo un estudio general en la literatura de las distintas formas de definir y entender el paradigma. En base a este análisis y sobre la definición de un programa orientado a objetos como un conjunto de objetos que interactúan enviándose mensajes, definimos lo que consideramos la esencia del paradigma. Lo caracterizamos como un conjunto de elementos básicos que operan con un conjunto de mecanismos básicos, y esta manipulación exhibe un conjunto de características básicas. Los elementos son objetos y mensajes, los mecanismos son pasaje de mensajes, creación y modificación. Las características básicas son identidad, polimorfismo, encapsulamiento y posibilidad de compartir comportamiento (sharing).

La capacidad de compartir comportamiento, tanto mediante clases con herencia, como mediante prototipos con delegación, es provista como primitiva por los distintos lenguajes de programación. Una de las características distintivas de cada lenguaje es la forma en la que provee el sharing. En nuestra conceptualización del paradigma, el sharing es una característica construible sobre otras. Analizamos este concepto en profundidad y encontramos una caracterización que permite clasificar y describir tanto las formas en que se presenta el sharing en los diferentes lenguajes existentes, como otras posibles formas de sharing no exploradas.

Estudiamos distintos formalismos propuestos para representar el paradigma, analizando su adecuación al modelo minimal buscado. Entre estos formalismos, los que más se adecúan a un modelo con sólo objetos y mensajes son los cálculos de objetos, en general desarrollados para estudiar sistemas de tipos. Entre estos cálculos, elegimos el impς-cálculo de Abadi y Cardelli, un cálculo imperativo no tipado, como base para nuestro trabajo.

Estudiamos la representación de clases que hicieron Abadi y Cardelli. Nos propusimos representar todos los mecanismos de sharing por objeto, tales como se encuentran en los entornos de prototipos, usando este cálculo como base. Llegamos a la conclusión de que para representar algunos mecanismos de sharing es necesario dotar a los objetos en el cálculo de la capacidad de ser extendidos con nuevos mensajes, por lo cual el cálculo de Abadi y Cardelli no resulta lo suficientemente expresivo. Diseñamos otro cálculo, impEς-cálculo, basado en el impς-cálculo, con una semántica muy parecida, y dos operadores para extender objetos: extensión conservativa y extensión no conservativa.

Definimos un pequeño lenguaje de usuario que llamamos Asterix. Este lenguaje tiene construcciones primitivas para especificar todas las posibles variaciones del sharing en un entorno de prototipos. Usamos las construcciones Asterix como especificación, y las implementamos usando el impEς-cálculo. De esta manera comprobamos que el impEς-cálculo es suficientemente expresivo para representar de manera sencilla en intuitiva todos los mecanismos de sharing por objetos, y podría utilizarse para dar semántica rigurosa a lenguajes de programación reales.

La delegación implícita es el más complejo de los mecanismos de sharing por objeto. Fue la implementación de la delegación implícita combinada con la definición de comportamiento propio de los objetos que delegan la que motivó la incorporación de los operadores de extensión.

La representación de los mecanismos de sharing propuesta permite explicarlos en base a los conceptos más básicos del paradigma. Además, esta representación es modular, permitiendo componer fácilmente distintas formas de sharing, y estudiar sus combinaciones. Dado que los mecanismos de sharing por grupo también se pueden representar en el cálculo, se puede combinar sharing por objeto y por grupo dentro del mismo marco. Esto permitiría estudiar sus interacciones y proveer un entorno en el cual un sistema pueda evolucionar de un modelo a otro a medida que progresa su desarrollo.

Estudiamos en profundidad la relación de nuestro trabajo con otros trabajos del área. Simultáneamente con nuestro desarrollo, Luigi Liquori definió un cálculo funcional con extensión de objetos basado en el ς-cálculo. Viviana Bono y Kathleen Fisher crearon un cálculo imperativo con extensiones. A diferencia de nuestro operador de extensión, el definido por ellas es funcional, es decir que un objeto extendido pierde su identidad. Estos dos trabajos están centrados en el diseño de sistemas de tipos para estos cálculos. Mens, De Volder y Mens diseñaron un cálculo que resuelve los conflictos entre la delegación y el encapsulamiento. Definimos los operadores presentados en ese cálculo en base al impEς-cálculo.

Comenzamos a estudiar el problema de definir una noción de equivalencia de comportamiento para objetos del cálculo, planteamos algunas ideas para tratar el problema, que queda como trabajo futuro.

Así como el sharing, otras características importantes de la programación Orientada a Objetos podrían ser estudiadas con un desarrollo similar al de este trabajo. Una de estas características es el encapsulamiento. La mayoría de los cálculos de objetos no representan encapsulamiento con el objetivo de simplificar su semántica. Una posible continuación de este trabajo sería estudiar como representar el encapsulamiento dentro del modelo propuesto. Dado que pudimos traducir a nuestro cálculo un cálculo que ataca el problema del encapsulamiento, hay buenas razones para creer que esta representación es posible. El lenguaje Asterix presentado en este trabajo provee una forma de encapsulamiento prohibiendo sintácticamente que el objeto sea modificado desde el exterior, pero no permite ocultamiento de mensajes privados.

Otra continuación para el trabajo es definir un sistema de tipos para el impEς-cálculo.

Como conclusión final de este trabajo podemos decir que los elementos y mecanismos básicos considerados son suficientemente poderosos para representar la capacidad de los objetos de compartir comportamiento, que es una característica esencial del paradigma.

Referencias

Referencias

[ABPPPR97] Argañaraz V., Baum G., Pons C., Presso M.J., Prieto M., Romero N., Formalizing Sharing Constructions in tha Object Oriented Paradigm. Anales de CACIC97. October, 1997.

[ABPPR98] Argañaraz V., Baum G., Presso M., Prieto M., Romero N., Simulation of Behaviour and Object Substitutability, Workshop on Precise Behavioral Semantics (with an emphasis in Business Specification), ECOOP 98. July, 1998.

[AC96] M. Abadi and L. Cardelli, A Theory of Objects, Monographs in Computer Science, Springer. 1996.

[APR98] Argañaraz V., Presso M.J, Romero N., A formal implementation of delegation using objects and messages. En 1998 ACM Student Research Poster Competition. February, 1998.

[AR90] Pierre America y Jan Rutten, A Layered Semantics for a Parallel Object-Oriented Languaje, Foundations of Object Oriented Languajes, Springer Verlag, 1990.

[BF98] V. Bono, K. Fisher, An imperative, first order calculus with object extension, Proc 5th Annual FOOL Workshop, pages 8.1-8.13. January, 1998.

[Blas94] Günther Blaschek, Objec-Oriented Programming with Prototypes, Springer-Verlag, 1994.

[BPF97] Kim B. Bruce, Leaf Petersen, Brian Fiech, Subtyping is no a good "match" for Object-Oriented Languages. Lectures Notes in Computer Science 1241. ECOOP'97, 11th European Conference, Finland. Springer, pags. 104-127. 1997.

[Bru93] Kim B. Bruce, Safe type checking in a statically typed Object-Oriented programming language, Proceedings ACM Symp. On Princples of Programming Languages, pags. 285-298. 1993.

[BSG95] Kim B. Bruce, Angela Schuett and Robert van Gent, PolyToil: a type safe polimorphic Object-Oriented language (extended abstract). Lectures Notes in Computer Science 952. ECOOP'95, 9th European Conference, Denmark. Springer, pags. 27-51. 1995.

[BW] Kim B. Bruce, Peter Wegner, An algebraic model of subtype and inheritance.

[BZ89] Ruth Breu, Elena Zucca. LNCS 405 – Fundations of Softaware Technology and Theoretical Computer Science. Proceedings. Nineth Conference, Bongalore, India. December, 1989.

[CP89] William Cook, Jens Palsberg, A denotational semantics of inheritance and its correctness. OOPSLA'89 Proceedings, pags 433 – 443. October, 1989.

[FM98] Kathleen Fisher, John C. Mitchell, On the relationship between classes, objects an data abstraction. Theory and Practice of Object Systems, Wiley Interscience Publication, John Wiley & Sons. January, 1998.

[FM95a] Kathleen Fisher, John C. Mitchell, A delegation-based object calculus with subtyping, Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95), pages 42-61, Springer LNCS 965. 1995.

[FM95b] Kathleen Fisher, John C. Mitchell, The development of type systems for Object Oriented languages, Theory and Practice of Object Systems, 1(3): 189-220. 1995.

[GR96] A. Gordon, G. Rees, Bisimilarity for a first-order calculus for objects with subtyping, Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages, 386-395. 1996.

[Lie86] Henry Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, Proceedings OOPSLA. 1986.

[Liq97] Luigi Liquori, An Extended Theory of Primitive Objects: First Order System, ECOOP'97.

[Mey97] B. Meyer, Object Oriented Software Construction, Second Edition, Phlaie. 1997.

[MHF94] John C. Mitchell, Furio Honsell, Kathleen Fisher. A lambda calculus of objects and method specialization. Nordic J. Computing ,1:3-37. 1994.

[MVM96] Kim Mens, Kris De Volder, Tom Mens. A Formalisation of Encapsulated Modification of Objects. Technical Report, Programming Technology Lab, Vrije Universiteit Brussel. June 1996.

[MW91] José Meseguer, Timothy Winkler, Parallel Programming in Maude. Proceedings of Research Directions in High Parallel Programming Languages, Sprigner-Verlag. June, 1991.

[SLU88] L.A. Stein, H. Lieberman, D. Ungar, A Shared View of Sharing: The Treaty of Orlando. In Object-0riented concepts, applications and databases, W. Kim and F. Lochowsky, eds., 31-48. Addison-Wesley. 1988.

[Ste87] Lynn Andrea Stein. Delegation is Inheritance. In OOPSLA '87. 1987.

[US91] David Ungar, Randall b. Smith. SELF: The Power of Simplicity. Lisp and Symbolic Computation: An International Journal, 4, 3, 1991.

[Wir86] Wirsing, M. "Structured algebraic specifications", in Theoretical Computer Science 43, 1986, pp 123-250

Apéndice

Función de Traducción de Asterix al Cálculo

Apéndice – Función de Traducción de Asterix al Cálculo

Top ≡

```
let
True = [ if = ς(x) x.then,
         then = ς(x) x.then,
         else = ς(x) x.else ] in
let
False = [ if = ς(x) x.else,
          then = ς(x) x.then,
          else = ς(x) x.else ] in
let
Pila_vacia = [ *agregar = ς(x) λ(rec) λ(elem) let ag = x.*agregar in
                 [ *borrar = ς(x) λ(rec) λ(elem)
                       let bool = (elem.=? (rec.dato)) in
                       let setThen = bool.then ⇐ ς(x) rec.next in
                       let setElse = bool.else ⇐ (ς(x) let del = rec.next.borrar(elem)
                                                       in rec.next ⇐ ς(x) del )
                       in bool.if,
                   *do = ς(x) λ(rec) λ(f) let a = rec.dato.propagate(f) in rec.tail.do(f),
                   *agregar = ς(x) λ(rec) ag(rec)
                   dato = ς(x) elem
                   tail = ς(x) rec
                   borrar = ς(x) x.*borrar(x)
                   do = ς(x) x.*do(x)
                   agregar = ς(x) x.agregar(x)
                 ]
               *borrar = ς(x) λ(rec) rec,
               *do = ς(x) λ(rec) λ(f) rec,
               agregar = ς(x) x.*agregar(x),
               borrar = ς(xn) x.borrar(xn),
               do = ς(xn) x.do(xn),
             ] in
let
Lista = [*agregar = ς(x) λ(rec) λ(elem) let nue = rec.pri.apilar(elem)
                                          in rec.pri ⇐ ς(xn)nue,
```

```
        *borrar = ς(x) λ(rec) λ(elem) let del = rec.pri.borrar(elem)
                                      in rec.pri ⇐ ς(xn)del,
        *do = ς(x) λ(rec) rec.pri.do,
        new = ς(x) let cv2 = Pila_vacia in  [pri = ς(xn) cv2,
                                             agregar = ς(xn) x.*agregar(xn),
                                             borrar = ς(xn) x.*borrar(xn) ,
                                             do = ς(xn) x.*do(xn) ,
                                             ]
        ] in

let ch = Lista.new in

[test = ς(self) False,
=? = ς(self) λ(other) let a = other.test ⇐ False
                      let b = self.test ⇐ True in other.test,

children = ς(self) ch,
register = ς(self) λ(child) self.children.agregar(child),
unregister = ς(self) λ(child) self.children.borrar(child),
propagate = ς(self) λ(f) let me = f(self) in self.children.do(f),
clone = ς(self) self.*clone(self),
delegate = ς(self) self.*delegate(self),
*delegate = ς(self)λ(rec) let d = rec.*delegate in
                          let c = rec.*clone in
                          let ch = Lista.new in
                          let newObj =
                              [ parent = rec,
                              delegate = ς(xd) xd.*delegate(xd),
                              *delegate = ς(xd) d,
                              clone = ς(xd) xd.*clone(xd),
                              *clone = ς(xd) c,
                              test = ς(self) False,
                              =? = ς(self) λ(other) let a = other.test ⇐ False in
                              let b = self.test ⇐ True in other.test,
                              children = ς(self) ch,
                              register = ς(self) λ(child) self.children.agregar(child),
                              unregister = ς(self) λ(child) self.children.borrar(child),
```

```
                    propagate = ς(self) λ(f) let me =
                                                f(self) in self.children.do(f)
                    ]
                in
                let registration = rec.register(newObj) in newObj
*clone = ς(self)λ(rec) let d = rec.*delegate in
                let c = rec.*clone in
                let ch = Lista.new in
                    [ delegate = ς(xd) xd.*delegate(xd),
                    *delegate = ς(xd) d,
                    clone = ς(xd) xd.*clone(xd),
                    *clone = ς(xd) c,
                    test = ς(self) False,
                    =? = ς(self) λ(other) let a = other.test ⇐ False in
                    let b = self.test ⇐ True in other.test,
                    children = ς(self) ch,
                    register = ς(self) λ(child) self.children.agregar(child),
                    unregister = ς(self) λ(child) self.children.borrar(child),
                    propagate = ς(self) λ(f) let me =
                                                f(self) in self.children.do(f) ]
]
```

TransProgram :

TransProgram o =

let Top =.... in trans (o)

Trans :

Trans (**object**

message1

message2

:

messagen

end)

=

(trans-⇐ **message1** (trans-⇐ **message2**. . . . . (trans-⇐ **messagen** (Top.clone))). . . . .)

Trans (**object**

parent: ***oParent***

message1

message2

:

messagen

end)

=

(trans-⇐ **message1** (trans-⇐ **message2**...(trans-⇐ **messagen** (Trans(***oParent***).delegate))...)

Trans (**object**

clone: ***oBase***

message1

message2

:

messagen

end)

=

(trans-⇐ **message1** (trans-⇐ **message2**...(trans-⇐ **messagen** (Trans(***oBase***).delegate))...)

Trans (o.m) = (Trans(*o*)) . m

trans-⇐

trans-⇐ (**message** m (arg_1,.., arg_n) x_1,..,x_m *b* **end**) translatedObject =
    let o = translatedObject in
    update o m
        (ς(self) λ(rec) λ(arg_1)..λ(arg_n) [x_1 : ς(bself) [ ]
                                                :
                                                x_m : ς(bself) [ ]
                                                val : ς(bself) (transBody rec *b*) {x_i←bself.x_i $^{i\varepsilon 1..m}$}].
                                                val)

trans-⇐ (**message** m **delegate in** *o1*) translatedObject =
    let o = translatedObject in
    let obj = Trans(*o1*) in
        update o m (ς(self) obj.*m)

trans-⇐ (**message** m **embed from** *o1*) translatedObject =
    let o = translatedObject in
    let mcode = (Trans(*o1*).*m) in
        update o m (ς(self) mcode)

trans-⇐ (**message** m **embed from parent**) translatedObject =
    let o = translatedObject in
    let mcode = translatedObject.parent.*m in
        update o m (ς(self) mcode)

trans-⇐ (**message** m **delegate in parent**) transaltedObject =
    let o = translatedObject in
        update o m ς(self) self.parent.*m

transBody :

transBody s (**self** m := (arg_1,.., arg_n) x_1,., x_m *b* **end**) =

extend s m

(ς(self) λ(rec) λ(arg_1)..λ(arg_n) [x_1 : ς(bself) [ ]

:

x_m : ς(bself) [ ]

val : ς(bself) (transBody rec *b*) {x_i←bself.x_i $^{i\varepsilon 1..m}$}

].val)

transBody s (**self** m := **delegate in** *obj*) =

let aDonor = Trans(*obj*) in

extend s m (ς(self) aDonor.*m)

transBody s (**self** m := **delegate in parent**) =

extend s m (ς(self) rec.parent.*m)

transBody s (**self** m := **embed from** *obj*) =

let obj = (Trans(*obj*).*m) in

extend s m (ς(self) obj)

transBody s (**self** m := **embed from parent**) =

let obj = s.parent.*m in

extend s m (ς(self)obj)

transBody s (**self changeParent:** *obj*) =

let newParent = Trans(*obj*) in

let unreg = s.parent.unregister(s) in

let reg = newParent.register(s) in

(s.parent ⇐ ς(self) newParent)

transBody s (**super** m) =

s.parent.*m(s)

transBody s (b m o_1 ... o_n) =

(...((trans(b).m (o_1)) (o_2)) .....)(o_n)

```
transBody s x := o =
        x<= ς(w) trans (o)

transBody s self = s

transBody s x = x

transBody s o1.o2 =
        let dummy = transbody s o1 in transbody s o2
```

---

update:

```
update obj m aMethod =
        let d = obj.*delegate in
            let c = obj.*clone in
                obj . m⇐ ς(x)x.*m(x)
                    .*m⇐ aMethod
                    .*delegate⇐ ς(x)λ(rec) ( d(rec) .m⇐ ς(z) z.*m(z)
                                                    .*m⇐ ς(z) z.parent.*m )
                    .*clone⇐ ς(x)λ(rec) ( c(rec) .m⇐ς(z) z.*m(z)
                                                .*m⇐ς(z) aMethod )
```

extend:

```
extend obj m aMethod =
   let extension = λ(extRec) let d = extRec.*delegate in
                                let c = extRec.*clone in
                               extRec . m⇐+ ς(x)x.*m(x)
                               .*m⇐+ aMethod
                               .*delegate⇐ ς(x)λ(rec)(d(rec).m⇐+ ς(z) z.*m(z)
                                                          .*m⇐+ς(z)z.parent.*m)
                               .*clone⇐ ς(x)λ(rec) ( c(rec) .m⇐+ ς(z) z.*m(z)
                                                          .*m⇐+ ς(z) aMethod )
   in
      let d = obj.*delegate in
      let c = obj.*clone in
            obj . m⇐ ς(x)x.*m(x)
               .*m⇐ aMethod
               .*delegate⇐ ς(x)λ(rec) ( d(rec) .m⇐ ς(z) z.*m(z)
                                                .*m⇐ ς(z) z.parent.*m )
               .*clone⇐ ς(x)λ(rec) ( c(rec) .m⇐ς(z) z.*m(z)
                                                .*m⇐ς(z) aMethod )
               .children.do (extension)
```