UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Una formalización del modelado multi-nivel

Autor: Mario Álvarez Picallo
Tutor: Juan de Lara Jaramillo

Julio 2014

Resumen

Los sistemas formales de modelado, como el **Unified Modeling Language** (**UML**), son bien conocidos por todos los ingenieros de software. Los mismos permiten razonar sobre la estructura a gran escala de los sistemas de la información, facilitando el análisis, diseño e implementación de los mismos, así como su mantenimiento posterior, y simplificando tareas como la refactorización automática o la generación de código.

Por otro lado, a pesar del éxito que han tenido estas herramientas, no están exentas de limitaciones. La más notable de las mismas es, quizás, que no se permite considerar más de un nivel de instanciación de una clase. Para solventar esta insuficiencia, existen extensiones, como el **Meta-Object Facility** (**MOF**), que aumentan esta funcionalidad añadiendo niveles adicionales entre los que se pueden establecer relaciones de instanciación.

El **metamodelado multinivel** o **metamodelado profundo** es un paradigma relativamente nuevo de modelado que elimina las restricciones de la estructura estándar de cuatro capas que propone el **MOF** permitiendo sistemas con un número arbitrario de niveles. Sin embargo, el entendimiento actual de esta herramienta está basado en nociones de teoría de categorías, una disciplina matemática que resulta difícil de tomar en relación a lenguajes de programación reales orientados a objetos.

Para comprender mejor el funcionamiento de esta nueva herramienta, poniéndola en contexto con la teoría existente, y dar pie a investigación futura, este trabajo pretende establecer una formalización de la misma basada en **teoría de tipos**, una rama de la teoría de la computación que estudia la semántica de las reglas de tipado en un lenguaje de programación o sistema similar. Con este objetivo, hemos construido y analizado dos sistemas formales que pretenden constituir la base del desarrollo de futuros lenguajes de programación basados en el metamodelado multinivel, y hemos desarrollado compiladores de los mismos a JavaScript para probar la aplicabilidad de estos modelos teóricos de manera práctica.

Palabras clave – metamodelado multinivel, metamodelado profundo, teoría de tipos, lenguajes de modelado, lenguajes de programación, ingeniería dirigida por modelos, cálculos de objetos.

Abstract

Formal modeling systems such as the **Unified Modeling Language** (**UML**), are widely used by software engineers. Such systems allow reasoning about the large-scale organization of information systems, easing their analysis, design and implementation, reducing maintenance costs and simplifying automated generation or refactoring of software.

On the other hand, despite the success that these tools have experienced, they are not devoid of limitations. Perhaps the most noticeable is the lack of support for more than one level of instantiation of a certain class. In order to solve this problem, extensions have been designed, such as the **Meta-Object Facility** (**MOF**), that expand the functionality of classical modeling frameworks allowing additional levels between which can be established instance-of relationships.

Multilevel metamodelling or **deep metamodelling** is a recently-developed modeling paradigm that aims to completely eliminate the restrictions inherent in the four-layer approach proposed by the **MOF**, replacing them with systems that allow for an unrestricted number of object layers. However, the current understanding of this tool is based on category theory, a mathematical abstraction that is hard to relate to real object-oriented programming.

In order to better understand the particularities of this new tool, putting it in context with the existing theory and facilitating subsequent research, this work intends to establish a formalization of said tool through the use of **type theory,** a branch of computer science that deals with the specification and semantics of typing rules for programming languages and similar formal systems. With this in mind, we've developed and analyzed two formal systems that intend to form the basis for the development of further programming languages based on the notions of multilevel metamodelling. Furthermore, we have developed compilers for said systems that show the practical applicability of these purely theoretical models.

Keywords – multilevel metamodelling, deep metamodelling, type theory, modelling languages, programming languages, model-driven engineering, object calculi.

Glosario

- **Modelado clásico**: cualquier paradigma de modelado que emplee instanciación superficial, habitualmente limitándose a modelos en dos capas de entidades que tienen exclusivamente faceta de tipo o de objeto.
- **Instanciación superficial**: una semántica de la relación de instanciación según la cual una entidad sólo puede determinar la estructura de sus instancias directas.
- **Clabject**: entidad de un modelo que posee tanto una faceta de tipo como una faceta de objeto.
- **Powertype**: patrón de diseño que permite definir clabjects en un lenguaje de modelado clásico limitado a dos niveles. También hace referencia a la entidad definida en dicho patrón.
- **Modelado multinivel**: un paradigma de modelado que estructura un modelo en una serie de capas adyacentes, entre las que se establecen relaciones de instanciación profunda.
- **Instanciación profunda**: una semántica de la relación de instanciación según la cual una entidad determina la estructura de sus instancias indirectas (instancias de instancias) mediante el mecanismo de la potencia.
- **Potencia**: atributo que se asocia a cada entidad (clabject o atributo) en un modelo multinivel y determina el nivel de instanciación al que dicha entidad se realiza.
- **Nivel**: atributo que se asocia a cada clabject de un modelo multinivel y determina la capa del modelo en el que este clabject reside.
- **Sistema formal/cálculo formal**: conjunto de reglas sintácticas y semánticas que codifican el comportamiento de un lenguaje o paradigma de programación.
- **Semántica operacional**: un método de asignar una semántica a un conjunto de estructuras sintácticas definiendo una relación de reducción entre ellas y reglas condicionales bajo las que se lleva a cabo dicha relación.
- **Sistema de tipos**: conjunto de reglas que asignan un tipo a algunas de las expresiones de un cálculo formal. Las expresiones a las que un sistema de tipos asigna un tipo se dicen bien tipadas.
- **Cálculo sigma**: un cálculo formal empleado para definir la semántica de un lenguaje orientado a objetos básico.
- **Cálculo sigma de primer orden**: una extensión del cálculo sigma con un sistema de tipos simple.
- **Cálculo alfa**: uno de los cálculos formales desarrollados en el presente trabajo, que extiende el cálculo sigma con las nociones de potencia e instanciación profunda.

- **Cálculo beta**: cálculo formal desarrollado en este trabajo, que extiende el cálculo beta con un sistema de tipos.
- **Arche**: un lenguaje simple basado en la semántica del cálculo alfa, que compila a código JavaScript.
- **Archetype**: un lenguaje simple basado en el cálculo beta, que compila a JavaScript.

Índices

Índice de contenidos

1. Introducción 1
  1.1. Motivación 1
  1.2. Objetivos 5
  1.3. Estructura de la memoria 6
2. Estado del arte 9
  2.1. Problemas de los enfoques clásicos 9
  2.2. Una solución parcial: *powertypes* 11
  2.3. El modelado multinivel 13
  2.4. Implementaciones actuales 18
3. Semántica formal 23
  3.1. Introducción 23
  3.2. Objetos sin clase: el cálculo sigma 24
  3.3. Sistemas de tipos 30
4. El cálculo multinivel 37
  4.1. Introducción 37
  4.2. Alfa: el cálculo multinivel no tipado 37
  4.3. Beta: añadiendo tipos al cálculo multinivel 44
5. Desarrollo de compiladores 53
  5.1. Motivación 53
  5.2. Tecnología empleada 53
  5.3. El lenguaje Arche y su compilador 54
  5.4. El lenguaje Archetype y su compilador 57
6. Conclusiones y trabajo futuro 61
  6.1. Conclusiones 61
  6.2. Trabajo futuro en cálculos multinivel 61
  6.3. Trabajo futuro en Arche/Archetype 62
  6.4. Aplicaciones potenciales 63
7. Bibliografía 65

Anexo I: Instalación de los compiladores ... 69
1. Prerrequisitos ... 69
2. Obteniendo el código ... 69
3. Compilando el código ... 70
4. Uso de los compiladores ... 70
Anexo II: Sintaxis de Arche ... 71
1. Analizador léxico ... 71
2. Analizador sintáctico ... 71
Anexo III: Sintaxis de Archetype ... 75
1. Analizador léxico ... 75
2. Analizador sintáctico ... 76

Índice de figuras

Figura 1. Ejemplo de modelo en dos capas....1
Figura 2. Modelando video como objeto....2
Figura 3. Modelando Video como clase....2
Figura 4. Una arquitectura de modelado en tres niveles....3
Figura 5. Modelo en tres capas con potencias....5
Figura 6. Metaclases Componente y Nodo....10
Figura 7. Definiendo instancias de las clases del metamodelo....11
Figura 8. Simulando más de dos niveles con powertypes....12
Figura 9. Ejemplo de instanciación profunda....14
Figura 10. Consecuencias de violar las restricciones sobre la potencia....15
Figura 11. Modelando Nodos y Componentes con potencias....16
Figura 12. Clase con un atributo dual....17
Figura 13. Desdoblando un campo dual en campos simples....17
Figura 14. Metaclases en Python....22
Figura 15. Jerarquía de clases en Smalltalk-80....22
Figura 16. Atributo impuestos presente en varios niveles del modelo....43
Figura 17. Estructura de clases del Listado 8....52

Índice de listados

Listado 1. Modelado en tres niveles con metaDepth (adaptado de (8)) 20
Listado 2. Extendiendo un modelo con un nuevo componente 21
Listado 3. Computando factoriales con el cálculo sigma 29
Listado 4. Ejemplo de programa en cálculo sigma de primer orden 35
Listado 5. Modelado multinivel en alfa 44
Listado 6. Incoherencia de tipos al sobreescribir métodos concretos. 49
Listado 7. Modelado multinivel en beta 51
Listado 8. Violación del metamodelado estricto 52
Listado 9. Implementación de un modelo multinivel en Arche 56
Listado 10. Implementación de un modelo multinivel en Archetype 58

Índice de tablas

Tabla 1. Sintaxis del cálculo sigma ... 25
Tabla 2. Variables libres en el cálculo sigma ... 25
Tabla 3. Sustitución de variables libres ... 26
Tabla 4. Sintaxis del cálculo sigma de primer orden ... 30
Tabla 5. Sintaxis de alfa ... 38
Tabla 6. Sintaxis de beta ... 46

1. Introducción

1.1. Motivación

El uso de herramientas y sistemas formales de modelado es muy común en la ingeniería del software. Estos métodos nos proporcionan un paradigma unificado con el que desarrollar distintas aplicaciones y sistemas de tecnologías de la información.

Las ventajas que los sistemas de modelado como UML (1) proporcionan son bien conocidas. Estos nos proporcionan un lenguaje unificado en el que generar diseños abstractos que luego pueden traducirse a código, permitiéndonos abstraernos de las particularidades de la tecnología subyacente que se decida utilizar. Además, proporcionan un modelo formal del código que permite realizar manipulaciones abstractas sobre el mismo, como refactorización automática o generación mecánica de código.

Sin embargo, una limitación importante del paradigma de modelado que nos ofrece el modelado clásico orientado a objetos es que considera únicamente modelos estructurados en dos niveles, es decir, modelos formados por un conjunto de clases que se relacionan entre sí por relaciones de herencia o asociación, y un conjunto de objetos que se relacionan con las clases por relaciones de instanciación.

En la *Figura 1* podemos ver un ejemplo de modelo clásico. En el mismo, las entidades se dividen claramente en dos capas, clases y objetos. Las clases se relacionan entre sí por relaciones de herencia y con los objetos por relaciones de instanciación. Los elementos de la capa superior tienen únicamente faceta de tipos, mientras que los de la capa inferior tienen a su vez faceta de instancia. Estas dos facetas permanecen claramente separadas en este enfoque.

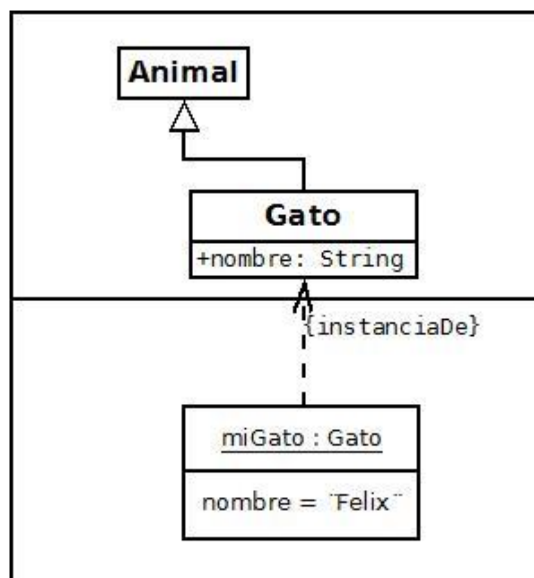


Figura 1. Ejemplo de modelo en dos capas

Un problema del mismo se hace aparente cuando consideramos ciertos problemas en los que parece necesario un tercer nivel de instanciación. Por ejemplo, consideremos un sistema en el que deseamos modelar un conjunto de productos de distintos tipos. Así, dos posibles modelos de este sistema quedan recogidos en los diagramas de las figuras *Figura 3* y *Figura 2*.

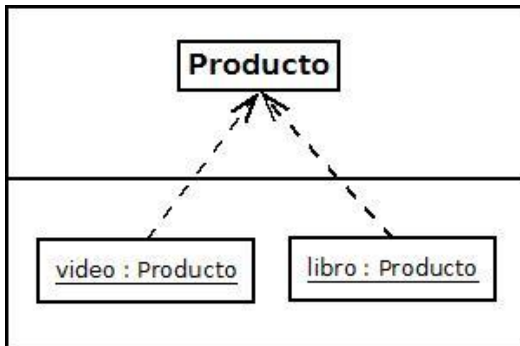


Figura 3. Modelando video como objeto

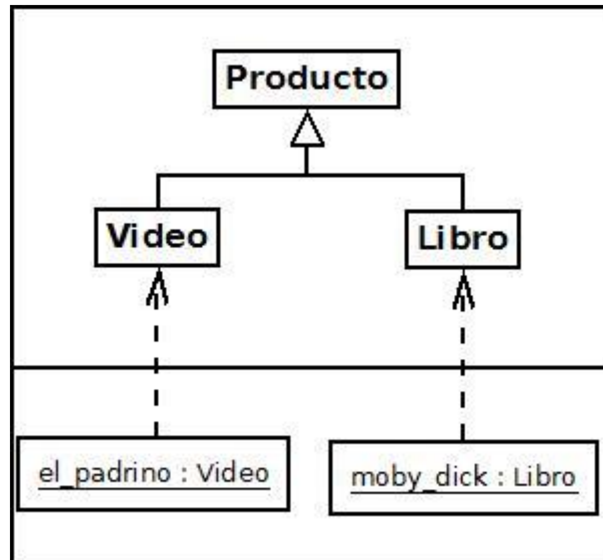


Figura 2. Modelando Video como clase

Cada uno de estos diagramas presenta una limitación distinta: el modelo expuesto en la *Figura 2* nos permite efectivamente considerar el vídeo `el_padrino` como una instancia de la clase `Video`, pero no nos permite considerar la misma como un objeto. Es decir, no podemos manipular las subclases de la clase `Producto` como si fueran objetos para, por ejemplo, crear nuevas instancias de `Producto` en tiempo de ejecución, o para escoger entre un tipo de producto u otro basándonos en el resultado de una operación. Por otro lado, el modelo que propone la *Figura 3* no nos permite considerar las entidades `video` y `libro`, instancias de la clase `Producto`, como clases. Es decir, este enfoque no nos permite instanciar `video` para obtener `el_padrino`. Esto se debe a que empleando modelado tradicional sólo podemos considerar un nivel de clases, que quedan instanciadas en un nivel de objetos.

Dependiendo del contexto, es decir, dependiendo de si la entidad *Video* en el dominio del problema se comporta como una clase o como un objeto, podemos emplear un enfoque u otro. Sin embargo, no nos es posible construir un modelo que refleje ambas facetas de la entidad. Por supuesto, en un ejemplo tan simple no parece necesario emplear más de dos niveles, pero en la práctica muchos modelos se descomponen de manera natural en varios niveles, al estar formados por varias capas de abstracción (2). Un ejemplo de esto, como se verá más adelante, surge al intentar formalizar sistemas de modelado como UML.

Para resolver este problema, podemos generalizar nuestro sistema de modelado para representar modelos formados por más de dos capas ontológicas. En un hipotético sistema de modelado que permitiese esta multiplicidad de niveles, sería posible plantear un modelo más apropiado, como el de la *Figura 4*.

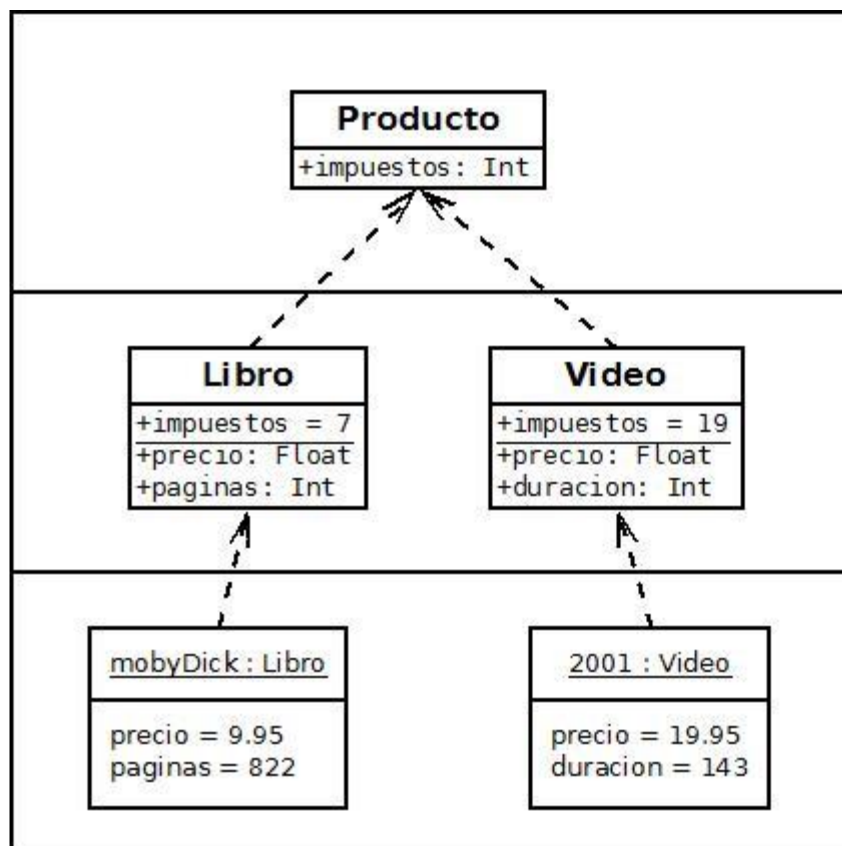


Figura 4. Una arquitectura de modelado en tres niveles

Para hacer posible esta clase de modelos, una de las extensiones más conocidas del modelo de dos capas, propuesta por el OMG, es el **Meta-Object Facility** o **MOF** (3). El MOF es un sistema de modelado originalmente destinado a constituir un sistema de tipos para el diseño de sistemas basados en CORBA. El mismo proporciona un meta-meta modelo estándar, el llamado nivel **M3**, que permite definir meta-modelos en el nivel **M2**. A su vez, los elementos del meta-modelo constituyen meta-clases, que están relacionadas con las entidades del modelo (el nivel **M1**) por una relación de instanciación análoga a la que se establece entre clases y objetos. A su vez, el nivel **M1** tipa el nivel **M0**, en el que residen los objetos del sistema que se está modelando, ya sean éstos entidades del mundo real u objetos del software que se está modelando.

En la práctica, sin embargo, la principal implementación del MOF (el **EMF**, **Eclipse Modeling Framework** (4), que constituye la implementación de referencia de facto del Essential MOF o EMOF), únicamente permite manipular dos de estos niveles: a saber, el nivel de meta-clases, que permiten definir lenguajes de modelado; y el nivel de objetos, formado por instancias de las meta-clases, que permiten definir modelos.

Sin embargo, limitarnos a enriquecer una arquitectura de modelado clásica con niveles adicionales

pone en relieve una serie de problemas que se derivan de la incapacidad de la relación de instanciación convencional de transmitir información a través de más de un nivel. Más adelante exploraremos estos problemas con más detalle, pero un ejemplo se puede apreciar ya en la *Figura 4*: un buen analista inmediatamente vería que el atributo `precio` aparece duplicado en las clases `Video` y `Libro`. La solución ideal sería elevar de algún modo el atributo `precio` a la clase `Producto`, pero nos es imposible transmitir este atributo desde dicha clase a las instancias de `Video` o `Libro`, que residen a dos niveles de distancia.

Para acabar con las limitaciones de este enfoque, se ha propuesto el metamodelado profundo (5), o metamodelado multinivel (6; 7). El mismo nos permite especificar modelos con cantidades arbitrarias de capas. Al igual que en el modelado clásico extendido con niveles adicionales, estas capas se relacionan entre sí por relaciones de instanciación, y cumplen papeles dobles como clases para la capa inferior y como objetos para la capa superior. Sin embargo, el modelado multinivel enriquece la relación de instanciación de modo que la misma permita transmitir información entre niveles separados.

Para aumentar la relación entre distintas capas del modelo, el metamodelado multinivel introduce la idea de **potencia** (6). La misma sirve para controlar la instanciación de un atributo de una clase en niveles sucesivos. De este modo, una instancia de una clase que posea un cierto atributo a potencia *M*, poseerá a su vez el mismo atributo a potencia *M – 1*. El atributo quedará realizado como un atributo real (un valor) cuando llegue a potencia *0*. Del mismo modo, una clase tiene asociada también una potencia que define cuántos niveles se puede instanciar, y un nivel, que define en qué capa del modelo se encuentra. Más adelante veremos que estos conceptos, sin ser idénticos, están estrechamente relacionados. En la *Figura 5* podemos ver una versión alternativa de la arquitectura propuesta en la *Figura 4*, que emplea potencias para elevar el atributo *precio* a la clase *Producto*. Entre corchetes se muestra la potencia de cada uno de los atributos. Como se puede observar, los atributos a nivel de clase de la *Figura 4* se convierten en atributos normales de potencia 0.

El metamodelado multinivel nos da un marco adecuado para definir modelos con múltiples niveles de “**Clabjects**” (amalgama de las palabras “class” y “object”), entidades, en los que cada nivel tipa al siguiente y queda tipado por el anterior. Existen ya una serie de estudios sobre estas ideas (6; 7; 5), y algunas implementaciones de lenguajes de modelado (8) o programación (7) basados en el mismo. Por desgracia, todos los lenguajes de programación orientados a objetos en uso en la actualidad siguen un paradigma de modelado clásico, en el que los sistemas se descomponen en objetos que son instancias de clases. El software en estos lenguajes no se presta al uso del metamodelado multinivel. Es necesario, pues, establecer y estudiar una nueva familia de lenguajes de programación que queden liberados de esta estructura de dos niveles.

Sin embargo, las perspectivas actuales sobre el metamodelado multinivel emplean el lenguaje de la teoría de categorías para definir sus conceptos (5). Las implementaciones existentes de lenguajes que emplean este formalismo son *ad-hoc* y no están basadas en un sistema de tipos formal (7). En contraste, los lenguajes basados en clases y objetos han sido ya representados como teorías de tipos (9), que permiten, por

un lado, estudiar matemáticamente el comportamiento de los programas escritos empleando este paradigma, y, por otro, servir de base para la implementación de compiladores, sistemas de refactorización automática, etcétera.

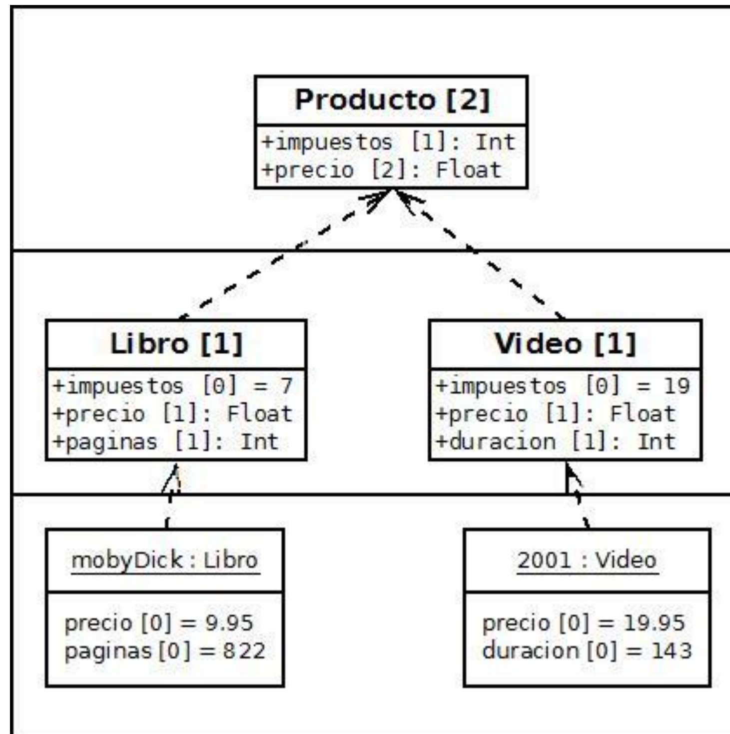


Figura 5. Modelo en tres capas con potencias

Para establecer una base común en la que se pueda asentar la investigación sobre sistemas y lenguajes basados en el metamodelado multinivel, este trabajo pretende definir una teoría de tipos en la que quede recogido el núcleo de su comportamiento. Esta teoría establecerá un lenguaje mínimo, más similar al cálculo lambda que a un lenguaje de programación real, junto con reglas para asignar tipos a las distintas expresiones del lenguaje.

Este trabajo nos permitirá ganar un mayor entendimiento del metamodelado multinivel y los patrones que se pueden definir mediante el mismo. Además esperamos adquirir perspectiva sobre el futuro de los lenguajes y herramientas basados en una estructura de tipos en varios niveles, y sentar las bases para el desarrollo de los mismos.

1.2. Objetivos

El principal objetivo de este trabajo es el desarrollo de sendos cálculos formales, en la tradición del cálculo lambda, que constituyan las bases formales de la semántica de lenguajes de programación basados en metamodelado multinivel: un cálculo no tipado al que denominaremos alfa, para definir la semántica operacional de los conceptos de potencia e instanciación profunda, y un cálculo tipado, beta, para asentar la semántica de las relaciones de instanciación y subtipado.

En el diseño y desarrollo de estos cálculos se tendrán en especial consideración los siguientes objetivos:

1. Recoger adecuadamente los aspectos fundamentales del metamodelado multinivel: ambos sistemas deben recoger la noción de clases a distos niveles, potencia de los atributos e instanciación profunda.

2. Simplicidad del sistema: los sistemas de tipos propuestos han de ser tan simples como sea posible, para facilitar su extensión posterior con nuevos conceptos así como su estudio formal.

3. Tratabilidad computacional: debe ser posible escribir un compilador que verifique la corrección de los programas escritos en dichos cálculos. Esto es especialmente importante para el cálculo beta, ya que es sencillo definir un sistema de tipos que, aunque sea aparentemente simple, no pueda ser verificado por un compilador.

4. Conexión con teorías existentes: en la medida de lo posible, se basará este sistema de tipos en otros sistemas de tipos existentes, principalmente el cálculo sigma no tipado y el cálculo sigma de primer orden, descritos ambos en la sección 3.

Como objetivo secundario, y a fin de asegurar que se han cumplido los criterios anteriores, se desarrollarán sendos compiladores de ejemplo que muestren la aplicabilidad real de los sistemas definidos. Se hace énfasis en que los mismos no son más que programas experimentales, cuyo único propósito es garantizar la implementabilidad práctica de los cálculos desarrollados.

Para facilitar la experimentación, se hará énfasis en una arquitectura extensible para el compilador, de modo que sea sencillo introducir nuevas primitivas y conceptos al lenguaje. Además, de cara a esta extensión, la estructura del compilador será lo más simple posible, omitiendo posibles optimizaciones o manipulaciones avanzadas del código, en detrimento, si es necesario, del código generado por el compilador.

1.3. Estructura de la memoria

Esta memoria se organiza en los siguientes apartados:

La presente sección 1 constituye una introducción a este trabajo, comprendiendo la motivación del mismo, así como los objetivos y prioridades que han guiado su desarrollo, junto con la presente sección en la que se esboza la estructura.

La sección 2 introduce una perspectiva general del estado del arte, comprendiendo cuatro sub-secciones: primero se identifican de manera concreta los síntomas de los métodos clásicos de modelado. Con esto en cuenta, se expone la teoría existente sobre el metamodelado multinivel, analizando sus distintas formulaciones. A continuación se muestran algunos de los distintos lenguajes de modelado o programación existentes actualmente basados en esta teoría, y se explora la aplicación informal de ciertos conceptos del metamodelado profundo en lenguajes de programación convencionales.

La sección 3 expone ejemplos de sistemas de tipos que han sido usados para formalizar lenguajes de programación convencionales, sirviendo este apartado también como una introducción a la nomenclatura y particularidades presentes en la definición de un sistema de tipos.

La sección 4 presenta el diseño de los cálculos que se han desarrollado durante el trabajo, incluyendo su semántica dinámica (reglas de ejecución) y estática (comprobaciones de tipos). Se señalarán además las similitudes o diferencias con sistemas de tipos existentes, y las diferentes decisiones que se han tomado en su desarrollo, junto con su motivación y alternativas. También se presentan codificaciones en estos sistemas de programas de ejemplo.

La sección 5 explica los detalles técnicos sobre el desarrollo de los compiladores de ambos lenguajes, su diseño e implementación, junto con la sintaxis concreta (externa), semántica y limitaciones de los mismos.

La sección 6 presenta las conclusiones a las que se han llegado durante el desarrollo del proyecto, y proporciona sugerencias para avenidas futuras de trabajo e investigación.

Por último, la sección 7 recoge las referencias bibliográficas consultadas durante el desarrollo de este trabajo.

2. Estado del arte

2.1. Problemas de los enfoques clásicos

Para entender las características particulares del modelado multinivel es necesario examinar las limitaciones del modelado que motivaron su creación. Con modelado clásico nos referimos a sistemas de modelado/metamodelado que estén constituidos por entidades (objetos/clases) distribuidas entre un cierto número de niveles, entre las que se da una relación de instanciación de la forma "X es una instancia de Y" o "X tiene tipo Y".

En estos sistemas clásicos la relación de instanciación sigue un modelo denominado **"instanciación superficial"** (6), basado en la premisa de que una entidad sólo puede condicionar la semántica de las entidades en el nivel directamente inferior, y no influencia en absoluto a las entidades de niveles inferiores. Este modelo viene motivado por el enfoque tradicional del desarrollo orientado a objetos, que está estructurado en dos niveles, a saber, clases y objetos. Además de esta limitación, en el enfoque denominado **"metamodelado estricto"** (10), que es el adoptado por UML y MOF y es el enfoque más habitual en la ingeniería dirigida por modelos, la única relación permitida entre dos entidades de distintos niveles es la relación de instanciación, y la misma sólo puede darse entre elementos de niveles adyacentes.

Debido a estas limitaciones, que emergen naturalmente al considerar el funcionamiento de los lenguajes de programación orientados a objetos, al intentar escalar la instanciación superficial con técnicas de metamodelado que empleen más de dos niveles de entidades (por ejemplo, la arquitectura de cuatro capas adoptada por el Meta-Object Facility) Atkinson y Kühne (6) han identificado problemas como el de la clasificación ambigua.

Para exponer este problema intentemos modelar la relación entre las entidades `Componente` y `Nodo` existente en UML, siguiendo el ejemplo descrito en (6). El estándar UML define un `Componente` como "una parte modular de un sistema que encapsula su contenido y cuya manifestación es reemplazable dentro de su entorno. Un componente define un comportamiento en términos de interfaces provistas e interfaces requeridas" (11). Por otro lado, un `Nodo` se define como "un recurso computacional sobre el cual artefactos pueden ser desplegados para su ejecución" (11).

En este modelo, las instancias de `Componente` representan tipos de componentes que pueden

formar el sistema que estamos modelando, como Servlet, o BaseDeDatos, y las instancias de Nodo constituyen tipos de recursos computacionales, como ServidorWeb. Este escenario ejemplifica, además, la necesidad de arquitecturas de modelado que reconocen más de dos niveles ontológicos de entidades

Entre las instancias de estos tipos existe una relación "reside en", conforme un servlet concreto es ejecutado por un servidor concreto. Esta relación está limitada por una relación correspondiente entre Componente y Nodo que es la que determina que un Servlet puede residir en un ServidorWeb. Para modelar esta relación entre instancias de instancias de Nodo e instancias de instancias de Componente es necesario introducir (meta) clases adicionales InstanciaNodo e InstanciaComponente para definir la relación "reside en" al nivel adecuado, tal y como se ilustra en la *Figura 6*.

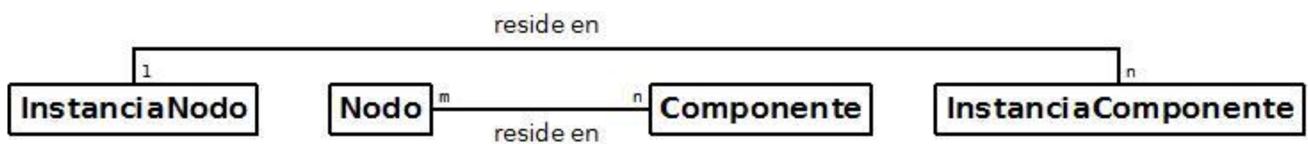


Figura 6. Metaclases Componente y Nodo

El problema que supone la instanciación superficial en este caso se hace evidente si consideramos Servlet y ServidorWeb, instancias de Componente y Nodo respectivamente. Al instanciar un Servlet queremos que el objeto resultante sea una instancia de la clase Servlet. Sin embargo, queremos también que esta instancia participe de la relación "reside en" que existe entre InstanciaNodo e InstanciaComponente, por lo que parece necesario que la misma sea una instancia de InstanciaNodo.

Esto plantea el dilema de si un servidor concreto (un elemento de nivel **M0**) debe ser una instancia de su clasificador correspondiente a nivel **M1** (Servidor) o a nivel **M2** (InstanciaNodo). Sin embargo, parece lógico que de estas dos relaciones la que estamos interesados en modelar es la que existe entre Servidor y MiServidor, y la relación entre MiServidor e InstanciaNodo existe únicamente para permitir que las entidades de nivel **M2** definan relaciones entre entidades de nivel **M0**. Este es el denominado problema de clasificación ambigua, que queda recogido gráficamente en la *Figura 7*.

En esta figura se puede apreciar claramente que las entidades MiServidor y MiServlet están sujetas a dos relaciones de instanciación que atienden a dos propósitos distintos: por un lado, la relación de instanciación que se establece entre MiServidor y Servidor es un ejemplo de instanciación ontológica, es decir, refleja la relación conceptual que existe entre un servidor concreto y la clase de todos los servidores. Por otra parte, la relación que se establece entre MiServidor e InstanciaNodo es de una

naturaleza distinta: existe únicamente para transmitir la relación “reside en” del nivel **M2** al nivel **M0**, existe únicamente para condicionar las características del objeto `MiServidor`, y no para reflejar una relación existente en la realidad. Este es un caso de instanciación lingüística, en contraste a la instanciación ontológica (12).

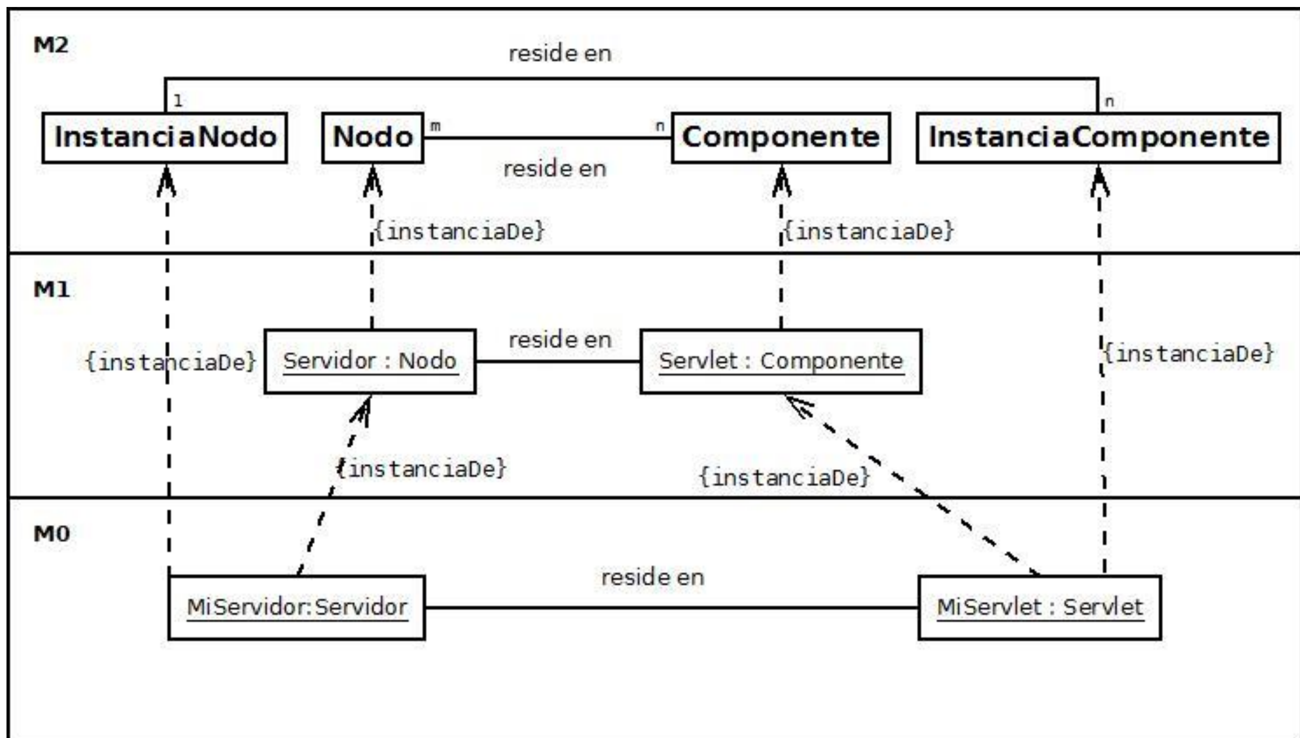


Figura 7. Definiendo instancias de las clases del metamodelo

Un problema adicional de la instanciación superficial es el de la replicación de conceptos (6; 13). El mismo tiene lugar debido a que el paradigma de instanciación superficial únicamente permite a un elemento influir en sus instancias directas, y se caracteriza por la aparición de una misma entidad en varios niveles del modelo. Un ejemplo es la duplicación del campo `precio` en ambas clases `Video` y `Libro` en el ejemplo de la *Figura 4*. Similarmente, en la *Figura 7*, podemos ver cómo la relación `reside en` aparece por duplicado debido a la pérdida de información que produce la instanciación superficial.

2.2. Una solución parcial: *powertypes*

Como hemos visto anteriormente, los sistemas de modelado que constan únicamente de entidades distribuidas en dos niveles ontológicos, a saber, un conjunto de clases y sus instancias, tienen problemas al expresar ciertos patrones (véanse las figuras *Figura 2* y *Figura 3*). Además, el modelo de instanciación superficial implica que añadir más niveles al modelo no es necesariamente la solución, ya que aparecen fenómenos como el problema de clasificación ambigua expuesto en la sección anterior.

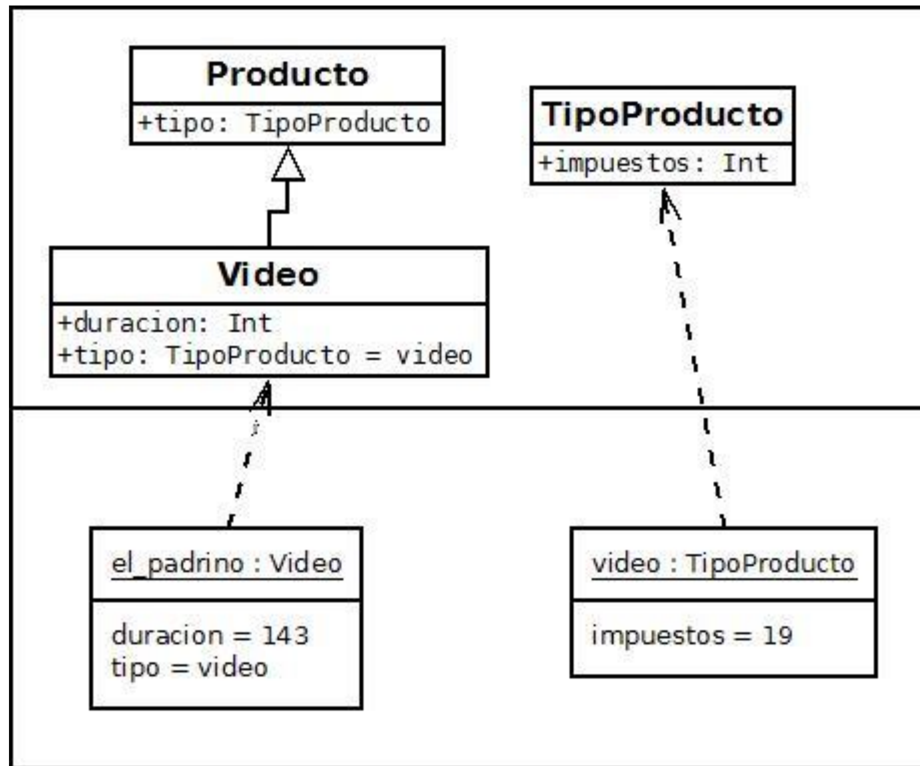


Figura 8. Simulando más de dos niveles con powertypes

Una manera de solucionar estos problemas es el patrón "***powertype***" (14; 15). El mismo desliga a una instancia de su clase, de modo que dicha clase pueda implementarse como una instancia. Este patrón convierte modelos 'profundos' de tres capas en modelos tradicionales de dos. En la figura inferior se puede ver un ejemplo de una traducción del modelo en tres capas de la *Figura 8* empleando el patrón powertype para reducir el modelo a dos niveles. En este caso, el patrón se aplica desdoblando la entidad `Video`, que en nuestro modelo anterior existía en un nivel intermedio, en una nueva clase `Video` y un objeto `video` que es a su vez instancia de la nueva clase `TipoProducto`. De este modo la naturaleza dual como clase y objeto de la entidad `Video` se preserva de la manera obvia: desdoblándola en dos entidades que reflejan cada uno de estos aspectos.

Este enfoque ha sido propuesto en la definición formal de procesos de desarrollo de software (16). Su principal ventaja es que no requiere grandes modificaciones sobre las metodologías de modelado actuales, y se puede implementar directamente en cualquier lenguaje moderno orientado a objetos. Sin embargo, en (6) y (16) se identifican algunos problemas del mismo: por un lado, requiere la introducción de nuevos elementos al modelo, elementos que no se corresponden claramente con entidades reales del dominio del problema (en el ejemplo anterior, la clase `TipoProducto` y su instancia `video`), requiriendo introducir complejidad adicional; por otro lado, la relación existente entre una clase y su objeto representante (la clase `Video` y el objeto `video` en el ejemplo) se basa únicamente en una convención, y no queda reflejada de

manera estricta en el modelo.

Sin embargo, el uso del patrón powertype exige ciertas modificaciones al modelado clásico: en concreto, la relación que se establece entre las facetas de clase y objeto (en la *Figura 8* la clase `Video` y el objeto `video`*)* no es compatible con las limitaciones del metamodelado estricto, que no permite relaciones entre entidades de niveles distintos.

Además, aunque los powertypes proporcionan una manera práctica de convertir modelos de tres niveles en modelos más convencionales de dos niveles, sin los problemas que causa la instanciación superficial, su aplicabilidad en el caso de modelos con más niveles ontológicos no se ha probado aún, y no es seguro que este patrón sea extensible a estos casos de manera práctica, es decir, sin introducir una complejidad adicional intolerable.

2.3. El modelado multinivel

Como respuesta a los problemas que presentan los sistemas de modelado que permiten únicamente dos niveles, y para evitar las complicaciones asociadas a un sistema con más de dos niveles pero con un modelo de instanciación superficial, como la replicación de conceptos o la clasificación ambigua, se ha propuesto el metamodelado multinivel, o metamodelado profundo (6). Este sistema de modelado nos permite estructurar nuestros modelos en cualquier número de niveles ontológicos.

La motivación del modelado multinivel surge de la consideración de que en un sistema de modelado con más de dos niveles, una entidad del modelo puede representar al tiempo un objeto y un tipo: el elemento constituye un objeto en tanto es una instancia de un elemento de un nivel superior, y es un tipo en tanto elementos de instancias inferiores están tipados por él.

Sin embargo, en el momento en el que una entidad *Y* puede ser instancia de una entidad *X* y, a su vez, verse instanciada en una entidad *Z*, podemos preguntarnos cuál es la relación que surge entre *X* y *Z*. Concretamente, podemos preguntarnos cómo contribuye la estructura de *X* a la de *Z*, es decir, si *X* puede definir características (campos, métodos, relaciones…) que deba heredar *Z* en tanto instancia de una instancia de *X*. Es de aquí de donde surgen las limitaciones impuestas por el modelo de instanciación superficial, ya que el mismo prohíbe necesariamente toda relación entre dos entidades *X* y *Z*. En este modelo de instanciación, los atributos de *Z* han de quedar únicamente determinados por la estructura que *Y* tiene en tanto clase, y no la que tiene como objeto (es decir, la heredada de *X*).

El modelado multinivel introduce como alternativa la **instanciación profunda** (6) o **caracterización**

profunda *(5)*. Empleando instanciación profunda, una clase puede determinar la estructura de sus instancias tanto directas como indirectas, a través de la noción de **potencia**. La potencia es un número entero asociado a cada elemento de un modelo. Este número nos indica la profundidad a la que el elemento correspondiente puede ser instanciado. Así, en la formulación de Atkinson y Kühne (6; 17), un elemento de potencia 0 corresponde a un elemento que no puede ser instanciado: un objeto, una clase abstracta, una interfaz o un atributo de un objeto. Los elementos de potencia 1 corresponden a su vez a clases o a métodos. Los elementos de potencia 2 y superiores no tienen análogos en modelado orientado a objetos, pero corresponden a entidades que pueden ser instanciadas dos veces. Es decir, la instanciación de un elemento de potencia 2 da lugar a un elemento de potencia 1.

Además de la potencia, el modelado multinivel introduce la noción estrechamente relacionada de **nivel**. Como el nombre indica, el nivel de un elemento del modelo es un número entero que representa el nivel ontológico del modelo en el que este elemento reside. De este modo, siguiendo el paradigma de la instanciación profunda, instanciar un elemento de nivel *n* y potencia *p* da lugar a un elemento de nivel *n-1* y potencia *p-1*. Esto implica que, si deseamos que el nivel y potencia resultantes sean válidas, sólo podremos instanciar elementos cuyo nivel y potencia sean mayores que cero (en caso contrario, el elemento resultante de la instanciación tendría un nivel o potencia negativos).

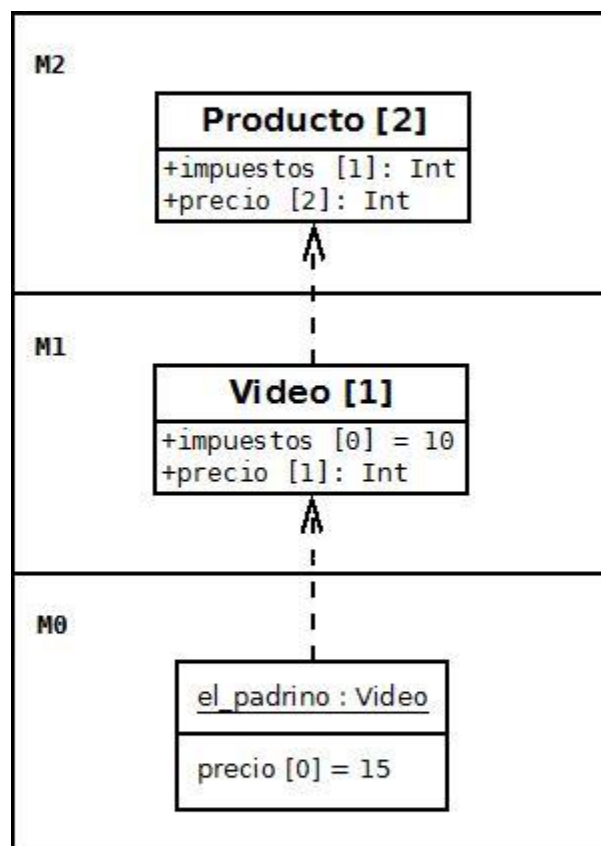


Figura 9. Ejemplo de instanciación profunda. Entre corchetes, la potencia de cada elemento

Una restricción adicional que surge de esta noción de instanciación y de nuestra definición de potencia es que la potencia de un elemento del modelo ha de ser forzosamente menor o igual que el nivel del mismo. Para entender por qué esta restricción es necesaria, basta considerar el caso de un elemento de nivel 0 y potencia 1. La potencia 1 nos indica que este elemento puede ser instanciado. Sin embargo, según las reglas de la instanciación profunda, sus instancias deberían tener nivel -1, como se muestra en la *Figura 10*.

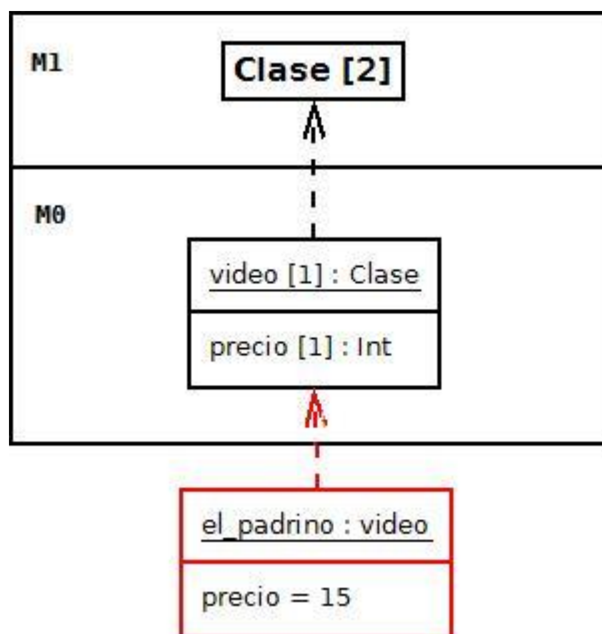


Figura 10. Consecuencias de violar las restricciones sobre la potencia

Esta restricción se aplica a que consideramos que el nivel **M0** es el nivel inferior de todo modelo. En principio podríamos generalizar la misma y considerar modelos con niveles arbitrarios, pero esto abre una serie de interrogantes sobre la naturaleza de las entidades en niveles inferiores. Además, si así lo deseamos, podemos imponer restricciones adicionales sobre la potencia y el nivel de un elemento. Por ejemplo, podemos establecer un límite al número de niveles que pueden contener nuestros modelos estableciendo que el nivel de todo elemento tiene que estar acotado por un cierto número *k*.

Empleando la instanciación profunda, estamos en posición de dar una versión más satisfactoria de modelos como los expuestos en la *Figura 7*, ya que no necesitamos añadir elementos adicionales al modelo para permitir transferir la información entre capas no adyacentes: podemos emplear para esto el mecanismo que nos ofrece la potencia. Un ejemplo simplificado se puede ver en la *Figura 11*. Nótese que al enriquecer la relación de instanciación ontológica desaparece la necesidad de añadir una relación de instanciación lingüística, y todas las relaciones de instanciación se establecen entre niveles adyacentes, por lo que no se violan los parámetros del metamodelado estricto.

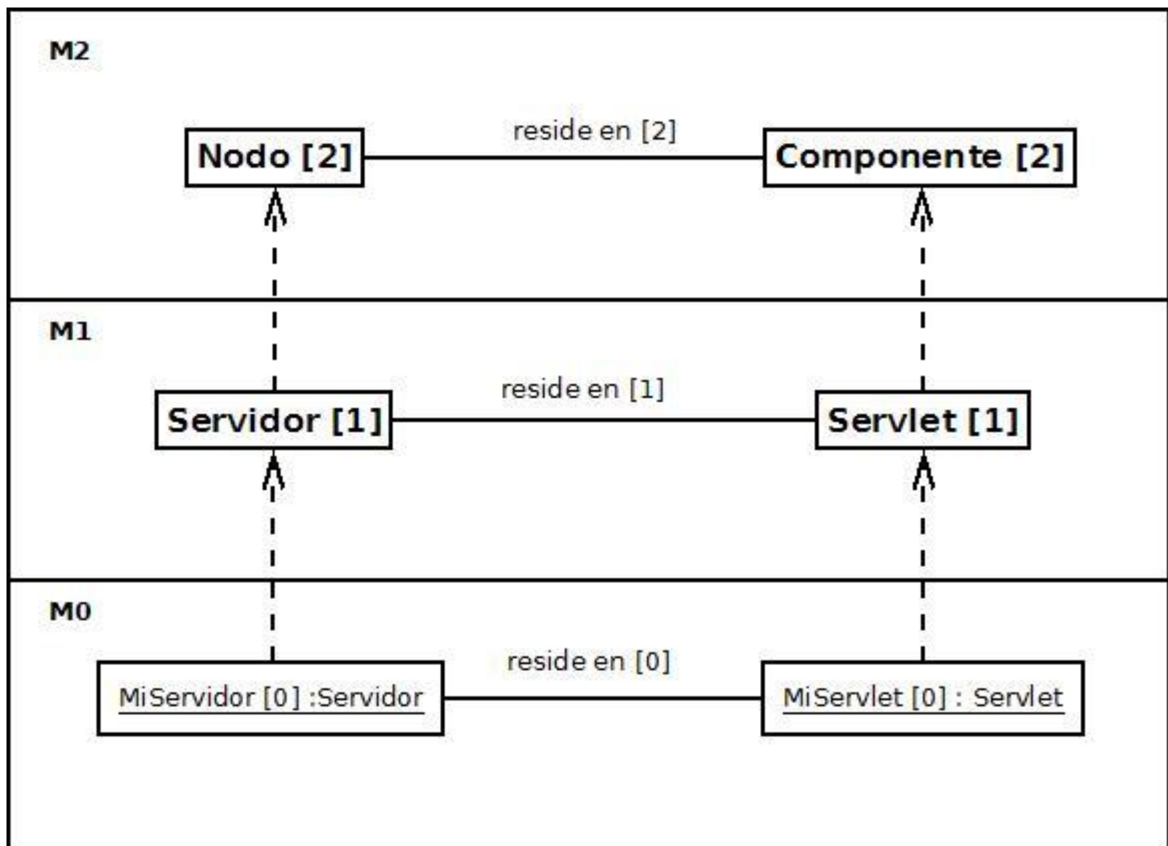


Figura 11. Modelando Nodos y Componentes con potencias

De este modo, este nuevo planteamiento no sólo simplifica el modelo, eliminando entidades innecesarias, sino que, además, acaba con el problema de la clasificación ambigua que aparecía en el caso anterior, y elimina posibles problemas de replicación de conceptos. Además, añadir niveles adicionales al modelo resulta trivial.

Además de las nociones de potencia y nivel, Kühne y Atkinson introducen otro concepto novedoso: los **campos duales**. Un campo dual es campo que tiene valor en varios niveles del modelo, en contraste a un atributo normal, que sólo tiene un valor cuando queda instanciado con potencia 0. De este modo, los campos duales tienen valor aunque tengan una potencia mayor que cero. No se debe confundir los campos duales con campos simples con valores por defecto: una clase puede definir un atributo de potencia 1 y asignarle un valor por defecto, pero el mismo no existe como atributo de la clase, sino como atributo de sus instancias. En contraste, un campo dual existiría como atributo de la clase y de sus instancias.

En la *Figura 12* se puede ver un ejemplo del uso de campos duales: la clase *Video* define el atributo *precio*, que existe como atributo estático propio de la clase, y también como un atributo propio de cada instancia de la misma.

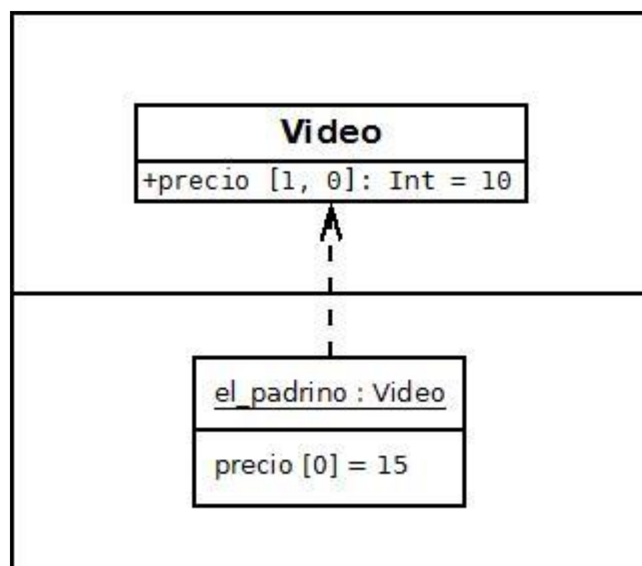


Figura 12. Clase con un atributo dual

Para entender mejor el significado de un campo dual de potencia *n*, podemos entenderlo como un conjunto de *n* campos de potencias *n, n-1, n-2…0* que comparten el mismo valor. De este modo, no es necesario introducir los campos duales como un concepto primitivo, sino que se pueden definir en términos de campos simples empleando el mecanismo de potencia. En la *Figura 13* podemos ver cómo este proceso de desdoblado se puede aplicar al ejemplo de la *Figura 12*.

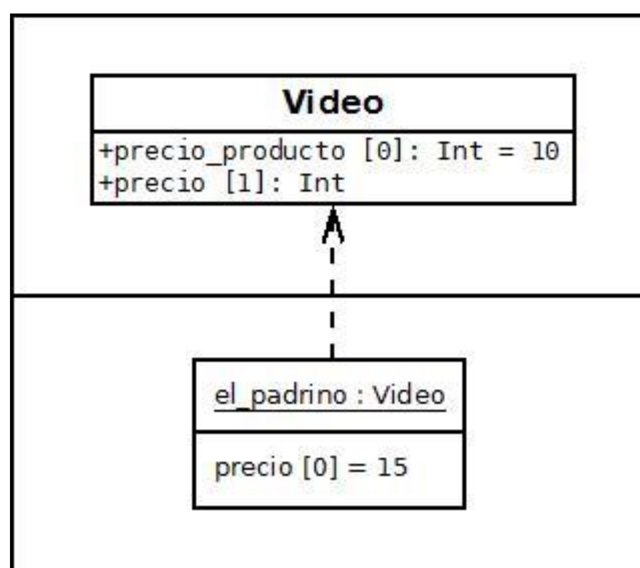


Figura 13. Desdoblando un campo dual en campos simples

Las ventajas del modelado profundo frente al uso de powertypes son claras. Al enriquecer la relación

de instanciación se pierde la necesidad de introducir las relaciones adicionales que son el síntoma de la clasificación ambigua, y el modelo gana en simplicidad al eliminarse las entidades auxiliares que necesitan los tipos potencia. Además, al contrario que el patrón powertype, el modelado multinivel escala a cualquier número de niveles de modelado.

Por otro lado, los powertypes se pueden emplear conjuntamente con enfoques de modelado y lenguajes de programación tradicionales, sin requerir cambios sustanciales en los mismos, lo cual puede resultar ventajoso. La traducción del patrón powertype a código Java es directa y simple, mientras que no es posible implementar modelos basados en instanciación profunda sin introducir mucha complejidad accidental. Esto se debe a que los lenguajes orientados a objetos actuales están firmemente basados en una estructura con dos niveles, no siéndonos posible definir niveles de entidades adicionales, y mucho menos aprovechar la flexibilidad de la noción de potencia para transmitir información entre niveles. El presente trabajo pretende constituir una alternativa, desarrollando un lenguaje en el que las técnicas de modelado profundo puedan ser implementadas directamente.

2.4. Implementaciones actuales

El modelado profundo es un paradigma relativamente nuevo y, como tal, no existe la misma base teórica de la que disfruta la orientación a objetos convencional. Sin embargo, se han desarrollado ya teorías y herramientas que se aprovechan de la misma de un modo u otro: bien para desarrollar lenguajes de programación (7) o bien para crear herramientas y lenguajes de modelado (8; 18).

El ejemplo más notable del primer campo es posiblemente el lenguaje de programación DeepJava (7), desarrollado por el propio Thomas Kühne junto con Daniel Schreiber. El mismo extiende el lenguaje de programación Java con soporte para definir clases en niveles superiores del modelo, es decir, meta-clases, meta-meta-clases y así sucesivamente, y permite enriquecer los atributos de las mismas con potencias, del modo prescrito por el modelado multinivel. Además de estas características, propias del modelado multinivel, DeepJava también está equipado con la capacidad de generar nuevas entidades (clases, meta-clases, etc.) en tiempo de ejecución, dinámicamente. Esta capacidad, tradicionalmente restringida a lenguajes con tipado dinámico, como Python (19) o Ruby (20), se obtiene a costa de sacrificar la seguridad del lenguaje, ya que es imposible, en tiempo de compilación, determinar si un objeto queda tipado por una clase que será generada dinámicamente.

Una limitación importante de DeepJava es que la única relación que puede haber entre entidades de distintos niveles del modelo es la de clasificación, es decir, aunque DeepJava elimina las restricciones impuestas por la instanciación superficial, sigue empleando metamodelado estricto. En términos prácticos,

esto se traduce en que una entidad de nivel *n* no puede tener atributos que referencien entidades de otros niveles. Sin embargo, es posible eliminar esta restricción para permitir más riqueza en los programas que se pueden definir, como veremos más adelante.

A más bajo nivel, DeepJava se ha implementado como una extensión conservativa del lenguaje Java, empleando el sistema Polyglot (21). Esto permite emplear código DeepJava junto con código Java puro, pero según los autores hace la sintaxis del lenguaje resultante algo aparatosa, al tener que ser compatible con el parser original de Java. Por otro lado, el entorno en tiempo de ejecución de Java no proporciona soporte para la generación dinámica de clases, por lo que los programas DeepJava necesitan incluir bibliotecas adicionales no estándar (22) que pueden perjudicar a la portabilidad de los programas resultantes.

Sin embargo, el mayor problema de DeepJava es quizás la ausencia de trabajo sobre el mismo: no hemos podido encontrar ningún recurso ni documentación relevante más allá del documento original (7), lo cual parece indicar que no está siendo activamente desarrollado. Esto hace que su utilidad como lenguaje de programación práctico sea cuando menos dudosa.

Un enfoque distinto es el de metaDepth (8), un lenguaje de modelado desarrollado en Java. En contraste con DeepJava, que es una extensión del lenguaje de programación Java, metaDepth es un lenguaje independiente que nos permite definir modelos con una potencia asociada, y los elementos que habitan dichos modelos. Siguiendo el paradigma del modelado profundo, metaDepth permite enriquecer los atributos de un elemento con una potencia para regular su instanciación.

Además, aunque se trata de un lenguaje de modelado y no de programación, metaDepth está integrado con la familia de lenguajes Epsilon (23), que permiten manipular elementos de los modelos definidos en metaDepth, efectivamente permitiendo la definición y ejecución de métodos sobre los mismos. Adicionalmente, se pueden definir restricciones y atributos derivados en las entidades de los modelos creados por el usuario, empleando para esto el lenguaje de programación Java o el Epsilon Object Language.

En el *Listado 1* se puede ver un ejemplo de modelado con metaDepth. Una peculiaridad de metaDepth es que la potencia de un elemento de un modelo es, por defecto, la potencia del modelo. Teniendo esto en cuenta, en este ejemplo, la potencia del elemento `TipoDeProducto` es 2, así como la potencia del atributo `precio`.

Otro aspecto interesante de metaDepth es que, junto a la instanciación estricta habitual en el modelado multinivel, permite también el uso de una relación de instanciación extensible. En modelos que emplean esta relación, se permite crear entidades en un cierto meta-nivel que no sean instancias de ninguna

entidad del meta-nivel inmediatamente superior, efectivamente permitiendo enriquecer el modelo de maneras no previstas en los niveles superiores, como se muestra en el **Error! Reference source not found.**. Esto resulta útil para introducir extensiones no previstas en **lenguajes específicos de dominio** (**DSLs**) (24) que consistan en más de un nivel: en estos lenguajes, el meta-nivel superior será habitualmente altamente genérico, necesitando de extensiones específicas en niveles inferiores y más especializados.

```
Model Tienda@2 {
      Node TipoDeProducto {
            VAT@1 : double = 7.5;
            price : double = 10;
      }
}

Tienda Libreria {
      TipoDeProducto Libro {
            VAT = 7;
      }
}

Libreria MiLibreria {
      Libro mobyDick {
            precio = 10;
      }
}
```

Listado 1. Modelado en tres niveles con metaDepth (adaptado de (8))

El ejemplo del *Listado 2* viola las limitaciones del metamodelado estricto, ya que introduce una entidad, *Autor*, y una relación, *escritor*, en el meta-nivel *Librería*, que no son instancias de ninguna entidad del meta-nivel *Tienda*. Sin embargo, metaDepth nos ofrece la capacidad de introducir nuevas entidades en niveles inferiores que no corresponden a instancias de entidades superiores.

Además de lenguajes explícitamente basados en el metamodelado multinivel, algunas de las ideas relativas al mismo están presentes en una forma u otra en otros lenguajes de programación. Muchos lenguajes dinámicos emplean metaclases, es decir, clases cuyas instancias son, a su vez, clases, para permitir al usuario extender el comportamiento de los objetos del lenguaje en maneras que la orientación a objetos habitual no permite. Las metaclases son elementos intrínsecamente relacionados con una arquitectura multinivel ya que son fundamentalmente incompatibles con el enfoque clásico de dos niveles.

Un ejemplo de metaclases se puede ver en el lenguaje de programación Python. La clase `type` (25) definida en este lenguaje es una metaclase, que el programador puede especializar para obtener nuevas metaclases. Al especializar la metaclase `type`, es posible redefinir la semántica de los elementos primitivos de la orientación a objetos como instanciación o llamada a métodos.

```
Model Tienda@2 {
    Node TipoDeProducto {
        VAT@1 : double = 7.5;
        price : double = 10;
    }
}

Tienda Libreria {
    TipoDeProducto Libro {
        VAT = 7;
        titulo : String;
        autor  : Autor;
    }
    Node Autor {
        nombre : String;
        libros : Libro[1..*]{unique};
    }
    Edge escritor(Libro.autor, Autor.libros) {
        anyo : int;
    }
}
```

Listado 2. Extendiendo un modelo con un nuevo componente

Una de las aplicaciones prácticas de las metaclases en Python es la implementación de clases abstractas, un concepto que no es primitivo en el lenguaje. Como se muestra en la

Figura 14, las clases abstractas como `Iterator` se crean instanciando la metaclase `abc.ABCMeta` (26), que es una subclase de la metaclase `type`. Además, en las versiones más actuales de Python, se han usado con éxito metaclases para permitir al usuario crear tipos enumerados o *enums* (27). Esto ilustra la potencia expresiva de las metaclases, que se pueden emplear para extender el lenguaje de formas novedosas.

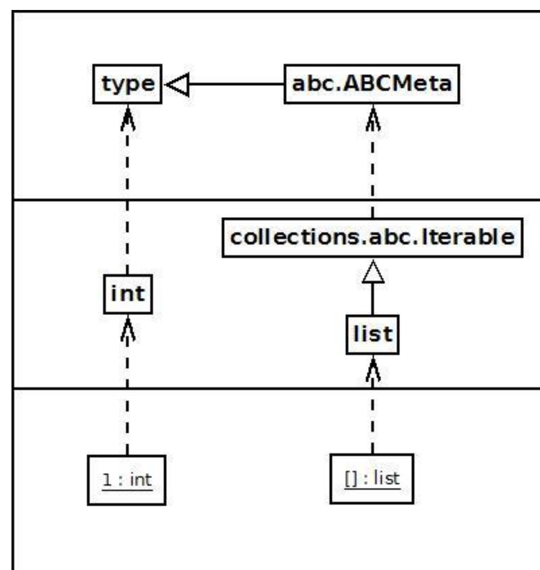


Figura 14. Metaclases en Python

El lenguaje de programación Smalltalk (28) emplea una arquitectura más sofisticada: en este lenguaje, toda clase es instancia de una metaclase particular, que es a su vez una instancia de la clase estándar `Metaclass`, tal y como se ve en la *Figura 15*. Este modelo representa una ruptura aún más radical con los paradigmas clásicos, ya que introduce una relación de instanciación circular entre la clase `Metaclass` y su metaclase `Metaclass class`, y no permite una separación en niveles independientes.

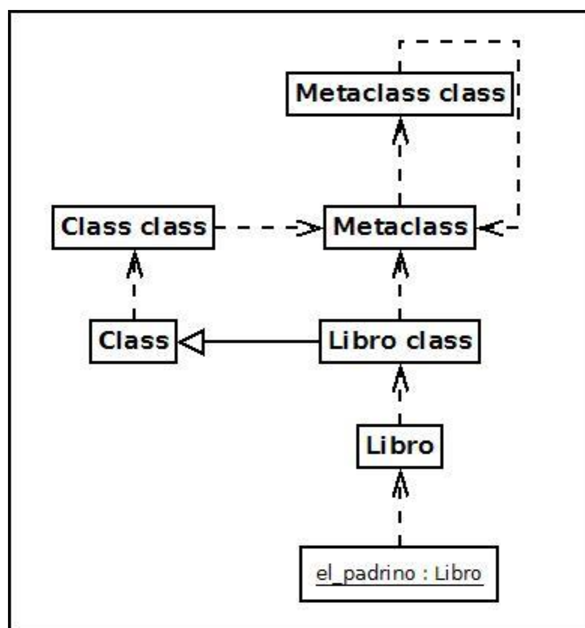


Figura 15. Jerarquía de clases en Smalltalk-80

Tanto Python como Smalltalk, junto con otros lenguajes con conceptos similares al de metaclases como Ruby (20), emplean tipado dinámico. Este es, en parte, el motivo por el que el uso de metaclases es posible en los mismos: las metaclases rompen el modelo de dos niveles (clases/tipos y objetos/valores) al que están fundamentalmente atados los lenguajes con tipado estático tradicional. El modelado multinivel puede servir para formar las bases de un lenguaje estáticamente tipado que permita el uso de metaclases para extender su propia arquitectura. La ventaja de este enfoque frente al uso de tipado dinámico es clara: un lenguaje con tipado estático detecta más errores en tiempo de compilación y genera código más legible, al actuar los tipos como una forma de documentación.

3. Semántica formal

3.1. Introducción

El objetivo de la semántica formal es entender el comportamiento de un lenguaje de programación de manera matemática. Para esto se emplean **cálculos formales**, objetos matemáticos que describen un lenguaje de programación concreto o, más comúnmente, un subconjunto simplificado de un lenguaje de programación. Esta sección pretende constituir una introducción superficial a la teoría de cálculos formales, haciendo énfasis en cálculos orientados a objetos. Una referencia más general sobre cálculos formales y su aplicación a la programación puede encontrarse en (29).

Un cálculo formal consta de tres componentes: una **sintaxis**, unas **reglas de reducción** y, en caso de que el cálculo sea tipado, unas **reglas de tipado**. La sintaxis de un cálculo formal describe precisamente la sintaxis de las expresiones del lenguaje que formaliza. Al tratarse de una abstracción matemática, y no de un lenguaje de programación real, la sintaxis de un cálculo formal emplea a menudo símbolos y notaciones muy distintas a las habituales en lenguajes de programación. Las reglas de reducción del cálculo definen el modelo de ejecución del lenguaje, es decir, cómo evaluar las expresiones para obtener resultados. Por último, las reglas de tipado de un cálculo formal definen el tipo que tiene cada expresión, y cuándo una expresión tiene un tipo correcto. La mayor parte de literatura escrita sobre cálculos formales se centra en las reglas de tipado, al ser en estas donde más riqueza y flexibilidad existe. Una referencia completa sobre sistemas de tipos, junto con ejemplos y teoremas relevantes, se puede encontrar en (30; 31).

El uso de cálculos formales para formalizar lenguajes y paradigmas de programación es muy antiguo, siendo el primer ejemplo el cálculo lambda, desarrollado por Alonzo Church en 1936 (32), que formaliza la idea de computación empleando únicamente funciones, dando lugar a la programación funcional. Este primer ejemplo era un lenguaje muy simple que no tenía siquiera reglas de tipado. Sin embargo, se ha demostrado que cualquier programa expresable en cualquier lenguaje de programación se puede escribir en el cálculo lambda. A partir del mismo aparecieron una serie de extensiones que enriquecían el lenguaje base con operaciones primitivas, o que añadían un sistema de tipos.

Mención especial merece el Sistema F (33), descubierto por Jean-Yves Girard en 1972 e, independientemente, por John Reynolds en 1974. Este cálculo, una extensión del cálculo lambda que introduce tipos paramétricos, constituye la base matemática de los tipos genéricos de Java y las plantillas de

C++ tal y como las conocemos hoy en día. Otras extensiones notables del cálculo lambda, así como una serie de resultados teóricos al respecto, pueden encontrarse en (32).

3.2. Objetos sin clase: el cálculo sigma

Para introducir la notación y conceptos involucrados en la definición de un cálculo formal simple, presentamos el **cálculo sigma** sin tipos. La formulación descrita aquí es una modificación menor de la descrita en (9).

El cálculo sigma es un cálculo formal que pretende capturar la esencia de la programación orientada a objetos. Las expresiones del cálculo sigma son totalmente orientadas a objetos, es decir, las únicas operaciones permitidas en el lenguaje son llamadas a métodos y actualizaciones de métodos, y el único tipo de valores que se permiten son objetos (esto es, el cálculo sigma no incluye la noción de tipos primitivos como enteros o booleanos). Para simplificar el lenguaje aún más, el mismo no establece una separación entre atributos y métodos: todos los campos de un objeto contienen métodos de cero argumentos, que pueden actuar como atributos o bien, con algo de código extra, como métodos más tradicionales de varios argumentos.

Asimismo, el cálculo sigma prescinde de las nociones de clases y herencia, siendo más similar al modelo de lenguajes como JavaScript/ECMAScript (34) o Self (35). La extensibilidad que en lenguajes más tradicionales se consigue mediante herencia de clases se puede implementar en el cálculo sigma creando una copia de un objeto ‘padre’ y extendiéndolo con los métodos deseados del objeto ‘hijo’, de una manera similar a la programación orientada a objetos basada en prototipos. Sin embargo, a diferencia de los lenguajes basados en prototipos, un objeto en sigma no tiene implícito un puntero a su prototipo que se emplee para resolver las llamadas a métodos que no existen en el objeto derivado. Si deseamos generar un nuevo objeto a partir de un cierto prototipo en sigma debemos crear una copia completa del mismo y añadirle nuevos métodos. Esto es muy ineficiente y limita la aplicación práctica del cálculo sigma como lenguaje de programación, pero este no es su propósito: el cálculo sigma, y los cálculos formales en general, pretenden ser únicamente herramientas de estudio y en modo alguno se proponen como lenguajes prácticos.

a, b, c ::=	Expresiones
x, y, z...	Variables
{**l**$_i$: sigma(**x**$_i$)**b**$_i$;} $(1 \leq i \leq n)$	Objeto literal
a.l	Selección de atributo
a.l <- sigma(**x**)**b**	Actualización de atributo
(**a**)	Expresión entre paréntesis

Tabla 1. Sintaxis del cálculo sigma

A partir de la especificación de la sintaxis de la *Tabla 1* podemos ver que se trata de un lenguaje muy simple, que incluye un conjunto muy reducido de conceptos. Una expresión sigma puede ser una variable (`x`, `y`, `z` o cualquier otro identificador válido), un objeto literal definido usando una sintaxis similar a JSON, o un acceso (lectura o escritura) de un atributo. En esta definición supondremos que las meta-variables **l**, $\mathbf{l_i}$ denotan nombres de métodos, y las meta-variables **x**, **y**, **z**, $\mathbf{x_i}$ denotan nombres de variables.

Por simplicidad, el cálculo sigma, que toma su nombre de la palabra clave que emplea para introducir un método, no tiene el concepto de atributo: todos los campos de un objeto contienen métodos de un único argumento (el argumento que contiene la referencia al propio objeto, `self` o `this` en otros lenguajes, es explícito en el cálculo sigma).

Antes de definir las reglas de ejecución, es necesario introducir dos conceptos auxiliares: el de **variables libres** y el de **sustitución**. Estos conceptos son aparentemente simples pero es necesario definirlos con cuidado.

Intuitivamente, las variables libres de una expresión son precisamente las variables que no están ligadas, es decir, las variables que no aparecen en el cuerpo de un atributo que las ligue a `this`. Definimos una función `FV` que asignará a cada expresión el conjunto de sus variables libres tal y como se especifica en la *Tabla 2*.

FV(**x**)	=	{**x**}
FV({$\mathbf{l_i}$: sigma($\mathbf{x_i}$)$\mathbf{b_i}$;})	=	∪(FV($\mathbf{b_i}$) - {$\mathbf{x_i}$})
FV(**a.l**)	=	FV(**a**)
FV(**a.l** <- sigma(x)**b**)	=	FV(**a**) ∪ (FV(**b**)- {**x**})
FV((**a**))	=	FV(**a**)

Tabla 2. Variables libres en el cálculo sigma

Estas reglas determinan unívocamente el conjunto de variables libres de una expresión del cálculo sigma. La noción de variable libre es integral al concepto de sustitución, que se definirá más adelante, ya que la sustitución tiene como objetivo sustituir las variables libres de una cierta expresión por expresiones complejas, dejando inalteradas las variables que no aparecen libres. Como veremos más adelante, hay que tener especial cuidado con la operación de sustitución, ya que su aplicación arbitraria conduce a comportamientos no deseados.

La sustitución captura la idea intuitiva presente en los lenguajes de programación de reemplazar el nombre de una variable por su valor correspondiente. Para definir la misma, emplearemos la sintaxis **a**[**x** -> **b**], que denota el resultado de sustituir la variable **x** por la expresión **b** en la expresión **a**. Las reglas que rigen esta operación se encuentran en la *Tabla 3*. La condición sobre la variable **x** en la última de las reglas puede parecer artificial. Sin embargo, eliminarla da lugar a incongruencias en el cálculo, como se verá más adelante. Si esta condición no se cumple, la sustitución, pues, no se puede realizar. Esto no es tan problemático como pudiera parecer, ya que es posible limitarse a renombrar la variable **x** en el método (sigma (**x**) **a**) de tal modo que la expresión resultante tenga la misma semántica pero **x** no aparezca libre en **b**, permitiendo así realizar la sustitución deseada.

Una vez definida la operación de sustitución podemos proceder a establecer la dinámica del cálculo sigma, es decir, el comportamiento de ejecución de los programas escritos en el mismo. Para establecer la dinámica de un cálculo o lenguaje de programación podemos decantarnos por varias opciones: podemos dar una **semántica denotacional** del cálculo, es decir, dar una traducción de las expresiones de nuestro lenguaje en las expresiones de un lenguaje conocido, de modo que la semántica del cálculo quede definida en términos de este lenguaje primitivo. Otra posibilidad es dar una **semántica ecuacional**, un conjunto de reglas que establecen cuándo dos expresiones ‘significan’ lo mismo, aunque sean formalmente distintas. Sin embargo, aquí nos vamos a decantar por representar la dinámica de este cálculo y los subsiguientes mediante una **semántica operacional**.

x[**y**->**b**]	=	**x**	
x[**x**->**b**]	=	**b**	
{$\mathbf{l_i}$: sigma($\mathbf{x_i}$)$\mathbf{b_i}$;}[**x**->**b**]	=	{$\mathbf{l_i}$: (sigma($\mathbf{x_i}$) $\mathbf{b_i}$)[**x**->**b**] ;}	
a.**l**[**x**->**b**]	=	(**a**[**x**-> **b**]).**l**	
(**a**.**l** <- sigma(**x**)**b**)[**y**->**c**]	=	(**a**[**y**->**c**]).**l** <- ((sigma(**x**)**b**)[**y**->**c**]))	
(**a**)[**x**->**b**]	=	(a[**x** -> **b**])	
(sigma(**x**)**a**)[**x**->**b**]	=	(sigma(**x**)**a**)	
(sigma(**x**)**a**)[**y**->**b**]	=	sigma(**x**)(**a**[**y**->**b**])	*si **x** no aparece en FV(**b**)*

Tabla 3. Sustitución de variables libres

Una semántica operacional describe el comportamiento dinámico de un lenguaje mediante reglas condicionales que establecen las condiciones bajo las cuales una cierta expresión ‘se reduce’ a otra. Esta relación de reducción que se establece entre expresiones se corresponde a la idea intuitiva de ejecutar un paso de un programa. Así, en la mayoría de lenguajes de programación, podemos decir que la expresión ‘1

`+ (2 - 3)`’ se reduce a la expresión ‘`1 + (-1)`’, que a su vez se reduce a la expresión ‘`0`’. Una introducción más general a la semántica formal de lenguajes de programación se puede encontrar en (36).

Para describir una semántica operacional, damos un conjunto de reglas de la forma ‘si una cierta expresión `a` se reduce a otra expresión `a’`, entonces la expresión `b` se reducirá a la expresión `b’`’. De nuevo ilustramos esto con un ejemplo que podría aplicarse a cualquier lenguaje de programación: ‘si `a+b` se reduce a `c`, y `f(x)` se reduce a `b`, entonces `a+f(x)` se reduce a `c`’. Esta notación no es estándar, y la usamos aquí por ser más simple y fácil de leer que la notación convencional empleada en textos clásicos (9; 29; 30; 31). Las diferencias son únicamente superficiales y la traducción entre ambas notaciones es un proceso mecánico.

La semántica operacional del cálculo sigma es especialmente simple, y sólo consta de tres reglas de reducción, que definimos aquí. Para ello introducimos la notación ‘`a ↣ b`’, que denotará el predicado ‘la expresión *a* se reduce a la expresión *b*’.

- Regla 1: objetos literales

Si
$\texttt{v = \{l}_{\texttt{i}}\texttt{: sigma(x}_{\texttt{i}}\texttt{)b}_{\texttt{i}}\texttt{;\}}$ *(1 ≤ i ≤ n)*
Entonces
`v ↣ v`

La intuición detrás de esta regla es clara: ejecutar un programa que se compone únicamente de un literal da como resultado el literal.

- Regla 2: selección de un atributo

Si
`a ↣ v’`
$\texttt{v' = \{l}_{\texttt{i}}\texttt{: sigma(x}_{\texttt{i}}\texttt{)b}_{\texttt{i}}\texttt{;\}}$ *(1 ≤ i ≤ n)*
$1 \leq j \leq n$
$\texttt{b}_{\texttt{j}}\texttt{[x}_{\texttt{j}}\texttt{->v']} \rightarrowtail \texttt{v}$
Entonces
$\texttt{a.l}_{\texttt{j}} \rightarrowtail \texttt{v}$

Esta es la regla más importante del cálculo sigma, y captura una noción esencial de la orientación a objetos. Intuitivamente, establece que invocar un método es equivalente a ejecutar el cuerpo del método con la variable `self` ligada a (sustituida por) el objeto al que

pertenece el método. Este es efectivamente el comportamiento que esperamos de cualquier lenguaje orientado a objetos, con la sutileza de que el cálculo sigma nos permite escoger qué nombre se va a ligar al objeto sobre el que se invoca el método, en vez de quedar definido por el lenguaje como las palabras clave `self` o `this`. Esto es similar al comportamiento de los métodos en Python (37), que reciben explícitamente la instancia del objeto sobre el que se invocan como primer argumento. Esto es fundamental para que el cálculo sigma constituya un modelo adecuado de un lenguaje de programación, ya que sin esta capacidad hay ciertos patrones que no podrían ser expresados (9).

- Regla 3: actualización de un atributo

Si
$a \rightarrowtail v'$
$v' = \{l_i: \text{sigma}(x_i)b_i;\}$ *(1 ≤ i ≤ n)*
Entonces
$a.l_j \text{ <- sigma}(x)b \rightarrowtail$
$\{l_j: \text{sigma}(x)b;\ l_i: \text{sigma}(x_i)b_i\ \ldots\}$

La regla 3 resulta directa al tener en cuenta la semántica de los atributos que queda definida en la regla 2. Tenemos que destacar que no se trata de una actualización en el sentido más puro de la palabra: ningún valor se altera, sino que se crea una copia nueva del objeto que se está actualizando, idéntica al original excepto por el valor del atributo a actualizar: si el mismo ya existía, su valor se sobrescribe. De no ser el caso, se crea un atributo nuevo. Alterar un objeto existente es imposible en el cálculo sigma, lo cual simplifica notablemente la definición de su semántica.

Habiendo definido estas reglas, se puede reducir cualquier expresión sigma a un valor, es decir, a una expresión que se reduce a sí misma por la regla 1. Sin embargo, es evidente que hay expresiones que no se pueden reducir, por ser erróneas. Un ejemplo simple podría ser la expresión (inválida) ‘`{}.x`’, que no se puede reducir por ninguna de las reglas dadas. A una expresión de este tipo, es decir, una expresión a la que no se puede aplicar ninguna regla, se la dice ‘**atascada**’ (‘*stuck*’ en inglés). Las expresiones atascadas en un cálculo formal corresponden aproximadamente a la idea de un error en tiempo de ejecución en un lenguaje de programación convencional.

Puede parecer que el cálculo sigma es insuficiente para modelar un lenguaje de programación orientado a objetos, ya que carece de muchas de las características que se esperan de un lenguaje de programación moderno, como tipos de datos básicos, funciones o constructores. Sin embargo, el cálculo

sigma es Turing-completo (38), es decir, cualquier algoritmo escrito en un lenguaje de programación se puede expresar en cálculo sigma, aunque no necesariamente con la misma eficiencia. La demostración escapa al propósito de este trabajo, pero se puede encontrar en (9). Como ejemplo del uso de este cálculo para definir programas, en el *Listado 3* presentamos una implementación de un método para calcular el factorial de un número, empleando una traducción orientada a objetos de los numerales de Church (39). Esta representación se basa en la idea de que se puede codificar el número n como una función que toma otra función como argumento y la aplica n veces a un cierto valor inicial. Aquí empleamos una codificación más sofisticada que asocia al número 0 la función $f_0(s, z) = z$ y al número n+1 la función $f_{n+1}(s,z) = s(f_{n+1},f_n(s, z))$, basada en el mismo concepto, pero proporcionando adicionalmente el número sobre el que se itera a la función iterada.

```
{
  zero: sigma(globals) {
    fold: sigma(zero) { result: sigma(foldArgs)foldArgs.onZero; };
    add: sigma(zero) { result: sigma(addArgs)addArgs.other; };
    mul: sigma(zero) { result: sigma(mulArgs)zero; };
  };
  succ: sigma(globals) {
    result: sigma(succArgs) {
      fold: sigma(succN) {
        result: sigma(foldArgs)
          ((foldArgs.onSucc.arg1 <- succN)
            .arg2 <- ((succArgs.prev.fold.onZero <- foldArgs.onZero)
                .onSucc <- foldArgs.onSucc).result).result;
      };
      add: sigma(succN) {
        result: sigma(addArgs)
          (globals.succ.prev <-
            (succArgs.prev.add.other <- addArgs.other).result)
            .result;
      };
      mul: sigma(succN) {
        result: sigma(mulArgs)
          (mulArgs.other.add.other <-
            (succArgs.prev.mul.other <- mulArgs.other).result)
          .result;
      };
    };
  };
  factorial: {
    result: sigma(factorialArgs)
      ((factorialArgs.arg.fold
        .onZero <- (globals.succ.prev <- globals.zero).result)
        .onSucc <- {
          result: sigma(self): (self.arg1.mul.other <- self.arg2).result;
        }).result;
  };
}
```

Listado 3. Computando factoriales con el cálculo sigma

3.3. Sistemas de tipos

En la sección anterior hemos visto un ejemplo de un cálculo formal muy simple: el cálculo sigma, que es adecuado para estudiar la semántica de un lenguaje dinámico orientado a objetos, como JavaScript. Este lenguaje no incluye muchos conceptos comunes en programación, como funciones o clases, pero los mismos se pueden emular en sigma, por lo que no es necesario introducirlos como primitivas.

Sin embargo, muchos lenguajes de programación están equipados con un sistema de tipos estático, es decir, una serie de reglas que determinan si una expresión es válida o no antes de ser ejecutada: las expresiones determinadas inválidas no tienen una semántica definida. El cálculo sigma no nos permite reflejar esta noción de corrección ya que define directamente la semántica de cada expresión sin dar un concepto de validez: una expresión será inválida si al reducirla (ejecutarla) se llega a una expresión atascada. La corrección de una expresión depende, en el cálculo sigma, de su comportamiento, y no de su forma, al contrario que una expresión en Java, cuya corrección es verificada por el compilador sin necesidad de ejecutarla.

A continuación extendemos el cálculo sigma con un sistema de tipos. Para ello se necesitan dos componentes: primero, es necesario extender la sintaxis del cálculo sigma con reglas sintácticas para formar tipos así como modificar las reglas existentes permitiendo la inserción de anotaciones de tipos. Además, se requieren reglas de tipado que determinarán bajo qué condiciones una cierta expresión tiene un tipo. Así, las expresiones a las que no se pueda asignar un tipo según estas reglas no serán válidas.

El cálculo que presentamos a continuación se denomina cálculo sigma de primer orden y extiende el cálculo sigma con tipos simples (es decir, no polimórficos). Este tiene la misma semántica que el cálculo sigma pero queda enriquecido con tipos estáticos que dan la garantía de que ninguna expresión **bien tipada**, es decir, ninguna expresión a la que se pueda asignar un tipo siguiendo las reglas de tipado, va a quedar atascada durante su ejecución.

A, B, C ::=		Tipos
	{$\mathbf{l_i}$: $\mathbf{A_i}$}	Tipo de objeto
a, b, c ::=		Expresiones
	x, y, z...	Variables
	{$\mathbf{l_i}$: sigma($\mathbf{x_i}$:$\mathbf{A_i}$)$\mathbf{b_i}$;} $(1 \le i \le n)$	Objeto literal
	a.l	Selección de atributo
	a.l <- sigma(**x:A**)**b**	Actualización de atributo
	(**a**)	Expresión entre paréntesis

Tabla 4. Sintaxis del cálculo sigma de primer orden

Como se muestra en la *Tabla 4*, el cálculo sigma de primer orden extiende la sintaxis del cálculo sigma no tipado con una nueva categoría sintáctica de tipos: los tipos son sintácticamente similares a objetos, con la diferencia de que en vez de asociar un método `sigma(`$\mathbf{x_i}$`)`$\mathbf{b_i}$ a cada nombre de atributo $\mathbf{l_i}$, asignan tipos $\mathbf{A_i}$. Además, los métodos en el cálculo sigma de primer orden indican en su declaración el tipo de la variable `self`, para hacer posible comprobar la corrección del cuerpo del método.

No es necesario definir explícitamente una semántica del cálculo sigma de primer orden: la sintaxis del mismo es idéntica a la del cálculo sigma sin tipos, a la que se han añadido anotaciones en la declaración de los parámetros de cada método. Para establecer una semántica de este cálculo, basta eliminar los parámetros de tipo y considerar la semántica de la expresión sigma resultante. Este proceso se denomina **borrado de tipos**.

Una vez establecida la base sintáctica del sistema de tipos, y quedando la semántica implícita mediante el borrado de tipos, se establecen las reglas de tipado. Las mismas tienen una forma similar a las reglas semánticas vistas en la sección 3.2, con el matiz de que introducen un contexto, tanto en sus hipótesis como en su conclusión. El contexto contiene los tipos de las variables libres que se conocen en el momento de comprobar el tipo de la expresión, ya que en general estos afectan a la comprobación de tipos. Como ejemplo, la expresión '`a + b`' en lenguaje Java tendrá tipo `int`, `float`, `long` u otro dependiendo del tipo que tengan las variables `a` y `b` en el contexto de la expresión.

Para presentar las reglas de tipado del cálculo sigma de primer orden vamos a introducir una serie de símbolos usados habitualmente en teoría de tipos. A un juicio de la forma 'la expresión `e` tiene tipo `T`' lo denotaremos por '`e : T`'. Las premisas y las consecuencias de nuestras reglas de tipado en general serán de la forma 'en un contexto en el que las expresiones a_i tengan tipos A_i respectivamente, la expresión `b` tendrá tipo `B`', que denotaremos con la sintaxis '`E, a1 : A1, …, an : An ⊢ b : B`'. Nótese que añadimos una meta-variable `E` al contexto para denotar que éste puede contener más información que la que necesitamos. Con esta notación, podemos expresar las reglas de tipado del cálculo sigma de primer orden.

- Regla 1: objetos literales

Si

$E,\ x_1 : A \vdash b_1 : B_1$

…

$E,\ x_n : A \vdash b_n : B_n$

Entonces

$E \vdash \{l_i\text{: sigma}(x_i{:}A)b_i;\} : A$

Esta regla establece que el tipo de un objeto queda determinado por los tipos de sus métodos. Es importante señalar la circularidad que se introduce en la misma, ya que al comprobar el tipo de cada método (las hipótesis de la regla) se asume que el parámetro `self` (denotado por x_i en cada método) tiene el tipo del objeto final. Nótese que esta regla no coincide exactamente con la sintaxis establecida para objetos literales en la tabla *Tabla 4*, ya que exige que todos los métodos asignen el mismo tipo a sus parámetros `self`.

- Regla 2: selección de un atributo

Si

$E \vdash a : \{l_i : B_i;\}$ *(1 ≤ i ≤ n)*

$1 \leq j \leq n$

Entonces

$E \vdash a.l_j : B_j$

Esta regla es quizás la más simple de las reglas de tipado y establece únicamente que si el objeto `x` tiene un atributo l_j de tipo b_j, entonces el acceso a ese atributo resultará en una expresión de tipo B_j.

- Regla 3: actualización de un atributo

Si

$E \vdash a : A$

$E \vdash a.l : B$

$E, x : A \vdash b : B$

Entonces

$E \vdash a.l \leftarrow \text{sigma}(x : A)b : A$

La regla 3 resulta también directa, pero introduce una limitación interesante al cálculo: sólo se permite actualizar un campo de un objeto con un método que tenga el mismo tipo, es decir, actualizar un objeto no puede alterar su tipo. Es interesante considerar que esta regla puede ser relajada para permitir actualizar un campo de un objeto con métodos de distinto tipo únicamente si la actualización no es destructiva, esto es, si crea un nuevo objeto actualizado en vez de alterar un objeto existente que pueda estar siendo referenciado desde otro punto del código.

Además, en esta formulación, no se permite añadir nuevos campos a un objeto existente. Esta limitación no es fundamental y se puede relajar, pero simplifica el planteamiento de esta

regla.

- Regla 4: extracción del contexto

Si **Entonces** `E, x : A ⊢ a : A`

Nótese que esta regla no tiene ninguna hipótesis, es decir, puede aplicarse siempre sin ninguna restricción sobre el contexto.

- Regla 5: extensión del contexto

Si
La variable x no aparece en el contexto E
`E ⊢ a : A`
Entonces
`E, x : B ⊢ a : A`

Estas dos últimas reglas son tautológicas y en muchos textos se dejan implícitas en vez de darse explícitamente. Las mismas, junto con otras reglas similares que no son necesarias al asumir que el contexto extra `E` es un conjunto arbitrario, forman lo que a veces se denominan **reglas estructurales**, y están presentes en la inmensa mayoría de sistemas de tipos. Sin embargo, existen sistemas de tipos que omiten intencionalmente alguna de las mismas (31; 40), los llamados **sistemas de tipos subestructurales**, que actualmente se emplean en algunos lenguajes de programación especializados como Rust (41).

Este sistema de tipos es primitivo, pero ya nos permite discriminar muchas expresiones inválidas como '`{}.foo`', a la que no se puede asignar un tipo. Sin embargo observemos que este sistema de tipos no es verdaderamente orientado a objetos. Si bien distintos lenguajes toman distintas definiciones de orientación a objetos (tipos estáticos o dinámicos, clases o prototipos, herencia simple o múltiple, etc…), un rasgo omnipresente, en una forma u otra, es la idea de polimorfismo. El planteamiento de la misma varía según los conceptos concretos que se apliquen a cada lenguaje, pero fundamentalmente se basa en la noción de que un objeto más específico (una instancia de una subclase, o un objeto con más métodos) puede, de manera transparente, tomar el papel de uno menos específico (una instancia de una superclase).

Al no tener las nociones de polimorfismo y subtipado en cuenta en el fragmento de sistema de tipos que hemos definido, nos encontramos con situaciones indeseadas. Por ejemplo, la expresión '`{foo:`

`(self : {foo : {};}) { bar: (self : {bar : {};}){}; };}`' no es válida, ya que el tipo del cuerpo del método foo resulta '`{ bar: {}; }`' pero, atendiendo a la anotación de su parámetro `self` el tipo que debería tener es '`{}`', que es un supertipo de '`{ bar: {}; }`' y, por tanto, debería ser compatible con la implementación proporcionada, a pesar de que nuestro sistema de tipos la rechace por no tener exactamente el mismo tipo.

Para resolver este problema, es necesario introducir, adicionalmente a la relación de tipado '`a : A`', una relación de subtipado de la forma 'el tipo `A` es un subtipo del tipo `B`', que denotaremos por '`A <: B`'. Las reglas que rigen esta relación en el cálculo sigma de primer orden son muy simples, y se detallan a continuación:

- Regla 1: reflexividad

Si
Entonces
`A <: A`

- Regla 2: transitividad

Si
`A <: B`
`B <: C`
Entonces
`A <: C`

- Regla 3: tipos de objetos

Si
$A = \{ l_i : A_i; \}$ *(1 ≤ i ≤ n)*
$B = \{ m_j : B_j; \}$ *(1 ≤ j ≤ m)*
Para todo j entre 1 y m, m_j corresponde a uno de los l_i y el tipo B_i es igual al tipo A_i correspondiente
Entonces
`A <: B`

Esta regla es la única que necesita explicación: intuitivamente, quiere decir que si un tipo `B` define un subconjunto de los métodos de un tipo `A` con los mismos tipos, entonces `A` es un subtipo de `B`.

- Regla 4: subsunción

```
Si
      B <: A
      E ⊢ b : B
Entonces
      E ⊢ b : A
```

Esta regla captura la esencia del polimorfismo, ya que es la que nos permite emplear una instancia de un subtipo como si fuera una instancia de un supertipo.

La introducción de tipos en este sistema tiene una consecuencia notable: se pierde la completitud del mismo, esto es, ya no se puede expresar cualquier algoritmo computable en el cálculo lambda. De hecho, no es posible traducir el *Listado 3* al cálculo sigma de primer orden, ya que el sistema de tipos del mismo no permite definir el tipo de los números naturales, que incluye un componente recursivo. Para emplear de manera práctica este cálculo formal es necesario enriquecerlo con tipos primitivos como Nat, el tipo de los números naturales, o Str, el tipo de cadenas de caracteres. Sin embargo, para ilustrar el uso de la sintaxis del mismo, presentamos un pequeño ejemplo en el *Listado 4*. Nótese especialmente la redundancia introducida en la declaración del tipo de cada parámetro `self` que puede evitarse introduciendo reglas de tipado más complejas que permitan a distintos métodos asignar un tipo distinto a sus respectivos parámetros.

```
{
  nombre: sigma(self: {nombre:Str;peso:Nat;bmi:Nat;altura:Nat;})
      ”Mario”;
  peso:   sigma(self: {nombre:Str;peso:Nat;bmi:Nat;altura:Nat;})
      80;
  altura: sigma(self: {nombre:Str;peso:Nat;bmi:Nat;altura:Nat;})
      180;
  imc:    sigma(self: {nombre:Str;peso:Nat;bmi:Nat;altura:Nat;})
      self.peso/self.altura.squared;
}
```

Listado 4. Ejemplo de programa en cálculo sigma de primer orden

En contraste a los sistemas de tipos de lenguajes más convencionales, como Java o C++, el sistema de tipos empleado aquí no hace necesario definir explícitamente los tipos o clases y sus relaciones de subtipado de antemano. Los tipos de cada expresión quedan implícitos por la forma de la misma, es decir, por sus métodos y los tipos de los mismos, y las relaciones de igualdad de tipos y subtipado se establecen en función de esta estructura, en vez de tener que ser explicitados por el programador. En el cálculo sigma de primer orden, dos tipos con la misma forma son fundamentalmente idénticos, en contraste con otros

lenguajes, que consideran distintos a dos tipos definidos bajo nombres distintos, aunque definan los mismos atributos y estructura.

A esta clase de sistemas de tipos basados en la forma de los tipos se denominan **sistemas de tipos estructurales** (30) (sin relación con los sistemas de tipos subestructurales mencionados anteriormente) y son habituales en el estudio de sistemas formales. Por contra, los sistemas de tipos basados en el nombre de un tipo se denominan **sistemas de tipos nominales** (30). Los sistemas de tipos estructurales no son comúnmente usados en el desarrollo de lenguajes de programación prácticos, aunque existen algunos lenguajes que emplean tipado estructural adicionalmente al tipado nominal. Notablemente, el sistema de objetos del lenguaje funcional OCaml emplea casi exclusivamente tipado estructural (42). Otro ejemplo práctico son las plantillas en el lenguaje de programación C++ (43), cuyos parámetros de tipo quedan restringidos por el uso que se hace de ellos en el cuerpo de la plantilla, pero no por el nombre de una clase o interfaz.

4. El cálculo multinivel

4.1. Introducción

En la sección anterior hemos descrito una formalización de algunos aspectos simples de la programación orientada a objetos. El cálculo sigma y sistemas similares (9) sirven de base formal para el desarrollo y estudio de lenguajes convencionales orientados a objetos. Sin embargo, estos sistemas son insuficientes para expresar los conceptos con los que trabaja el modelado multinivel. Esto causa una impedancia entre un modelo realizado empleando técnicas de metamodelado profundo y una implementación en software del mismo, que da lugar a la aparición de técnicas especializadas y poco escalables para evitarlo, como el patrón powertype descrito en 2.2.

Para resolver esto, es necesario el desarrollo de lenguajes de programación que se ajusten al paradigma del modelado multinivel, como (7), de modo que los patrones que emergen en modelos multinivel puedan expresarse de manera clara y directa en el software. En esta sección exponemos dos cálculos formales, uno no tipado y otro tipado, de manera análoga a los expuestos en las secciones 3.2 y 3.3, que extienden y modifican el cálculo sigma con las nociones necesarias para constituir un sustrato adecuado sobre el que construir lenguajes de programación en los que se puedan aplicar los patrones y conceptos del modelado multinivel con una dificultad mínima.

Estos cálculos pretenden servir por un lado como base formal para determinar los aspectos fundamentales de la semántica de futuros lenguajes de programación multinivel y, por otro lado, servir como banco de pruebas formal de cara a futura investigación sobre esta semántica, aportando un lenguaje común para el estudio sistemático de lenguajes multinivel.

4.2. Alfa: el cálculo multinivel no tipado

Hemos obtenido una primera aproximación a un cálculo multinivel extendiendo el cálculo sigma no tipado para reflejar la noción de potencia de un atributo. Este lenguaje nos sirve como una base simple para definir la semántica de la noción de potencia de un atributo, y supondrá la base de la semántica del cálculo multinivel tipado que desarrollamos a continuación.

Con este propósito, hemos extendido el cálculo sigma con la notación necesaria para asociar una

potencia a cada método de un objeto. En el enfoque adoptado aquí hemos decidido considerar el nivel y la potencia de una entidad, tal y como los definen la teoría del metamodelado multinivel, como parte del tipo de un objeto: el propósito de los mismos es establecer límites adicionales a la semántica de un objeto (en principio podríamos considerar instanciar un objeto de potencia cero, o un objeto de nivel cero con métodos de potencia mayor) por lo que, para maximizar el número de programas a los que se atribuye una semántica, no lo tenemos en cuenta para este lenguaje no tipado. Esto quiere decir que cualquier entidad en el sistema formal alfa es susceptible de ser instanciada, incluyendo entidades que en un lenguaje con niveles residirían en el nivel cero, resultando en objetos sin atributos. Una posible ventaja de este enfoque es que nos permitiría dar una semántica a una hipotética extensión del modelado multinivel en la que se permitan entidades de potencia *.

Esta liberalidad tiene como objetivo permitir el uso del cálculo alfa para definir la semántica de una variedad mayor de cálculos tipados más complejos. El resultado es un cálculo orientado a objetos que nos permite especificar objetos con atributos de distintas potencias e instanciar los mismos empleando instanciación profunda.

La principal diferencia respecto del cálculo sigma, aparte de la introducción de potencias, es la inclusión de un operador de instanciación explícito, ya que el cálculo sigma, al no tener ninguna noción de clase, no requiere de uno. Como se puede ver en la *Tabla 5*, la sintaxis de alfa es prácticamente idéntica a la del cálculo sigma.

a, b, c ::=		Expresiones
	x, y, z...	Variables
	{$\mathbf{l_i}$: sigma[$\mathbf{p_i}$]($\mathbf{x_i}$)$\mathbf{b_i}$;} *($1 \leq i \leq n, p_i > 0$)*	Objeto literal
	a.l	Selección de atributo
	a.l <- sigma[$\mathbf{p_j}$]($\mathbf{x}$)**b**	Actualización de atributo
	(**a**)	Expresión entre paréntesis
	new **a**	Instanciación

Tabla 5. Sintaxis de alfa

La semántica de alfa se basa también en la del cálculo sigma. Partiendo de las mismas nociones de variables libres y sustitución que se expusieron en la sección 3.2, basta introducir una nueva regla que rige el comportamiento de la instanciación, construcción que no estaba presente en el cálculo sigma, y modificar la regla que regula la lectura de un atributo para tener en cuenta la potencia del mismo.

- Regla 1: objetos literales

Si

```
v = {li: sigma[pi](xi)bi;}
```
(1 ≤ i ≤ n)

Entonces

```
v ↣ v
```

Esta regla es idéntica a su equivalente en el cálculo sigma sin tipos, aparte de una pequeña modificación para acomodar los nuevos elementos sintácticos introducidos por el cálculo sigma.

- Regla 2: selección de un atributo

Si

```
a ↣ v'
v' = {li: sigma[pi](xi)bi;}
```
(1 ≤ i ≤ n)

$1 \le j \le n$, $p_j = 0$

```
bj[xj->v'] ↣ v
```

Entonces

```
a.lj ↣ v
```

La única diferencia entre esta regla y su equivalente en el cálculo sigma es que no permitimos el acceso a métodos de potencia mayor que cero.

- Regla 3: actualización de un atributo

Si

```
a ↣ v′
v′ = {lᵢ: sigma[pᵢ](xᵢ)bᵢ;} (1 ≤ i ≤ n)
1 ≤ j ≤ n
```

Entonces

```
a.lⱼ <- sigma[p](xⱼ)b ↣
     {l₁: sigma[p₁](x)b; …
      lⱼ: sigma[p](x)b;
      …}
```

Si

```
a ↣ v′
v′ = {lᵢ: sigma[pᵢ](xᵢ)bᵢ;} (1 ≤ i ≤ n)
1 no está entre los lᵢ
```

Entonces

```
a.l <- sigma[p](x)b ↣
     {l₁: sigma[p₁](x₁)b; …
      l: sigma[p](x)b}
```

En este lenguaje, permitimos redefinir un atributo con cualquier potencia, incluyendo una potencia distinta de la original (siempre y cuando sea una potencia válida, es decir, un número natural). Es razonable exigir que la potencia del nuevo atributo sea la misma que la del atributo que se está redefiniendo, pero para hacer la semántica básica lo más flexible posible esta regla no impone ninguna restricción al respecto.

- Regla 4: instanciación de un objeto

Caso base:

Si

```
a ↣{}
```

Entonces

```
new a ↣ {}
```

Caso recursivo:

Si

```
a ↣ {lᵢ: sigma[pᵢ](xᵢ)bᵢ;} (1 ≤ i ≤ n)
Si   p₁ = 0
Entonces
     new a ↣ new {l₂: sigma[p₂](x₂)b₂; …}
Si   p₁ > 0
Entonces
     new a ↣
          new {l₂: sigma[p₂](x₂)b₂; …}.l1
               <- sigma[p₁-1](x₁)b₁
```

Nótese especialmente el carácter recursivo de la regla, que define primero la instanciación de

un objeto vacío para luego definir la instanciación de un objeto con n métodos en términos de la instanciación de un objeto con n-1 métodos al que se le añade un método adicional.

Estas reglas tan simples pueden ser extendidas fácilmente: por ejemplo, al añadir campos que existan en varios niveles del modelo, tal y como se muestra en la *Figura 16*. Para esto basta extender la sintaxis de la *Tabla 5* de modo que permita emplear un símbolo especial (aquí hemos empleado `*`) en vez de un número natural como potencia de un atributo, y añadir las siguientes modificaciones a las reglas de reducción anteriores:

- Regla 2a: selección de un atributo

Si

$v' = \{l_i: sigma[p_i](x_i)b_i;\}$ $(1 \leq i \leq n)$

$a \rightarrowtail v'$

$1 \leq j \leq n$

$p_j = 0$ o $p_j = *$

$b_j[x_j\text{->}v'] \rightarrowtail v$

Entonces

$a.l_j \rightarrowtail v$

- Regla 4a: instanciación de un objeto

```
Caso base:
Si
      a ↣ {}
Entonces:
      new a ↣ {}
Caso recursivo:
Si
      a ↣ {li: sigma[pi](xi)bi;} (1 ≤ i ≤ n)
      Si    p1 = 0
      Entonces
            new a ↣ new {l2: sigma[p2](x2)b2; …}
      Si    p1 > 0
      Entonces
            new a ↣ new {l2: sigma[p2](x2)b2; …}
                        .l1 <- sigma[p1-1](x1)b1
      Si    p1 = *
      Entonces
            new a ↣ new {l2: sigma[p2](x2)b2; …}
                        .l1 <- sigma[*](x1)b1
```

Otra extensión que podemos añadir surge de considerar la noción de métodos estáticos. Un método estático es un método que corresponde a una clase, pero al que se puede acceder desde sus instancias. El paradigma del metamodelado multinivel permite definir un método estático simplemente como un método de potencia 0 en un elemento del modelo de nivel 1. Sin embargo, una diferencia importante de este tratamiento de los métodos estáticos es que no es posible acceder a ellos desde una instancia de la clase, sino que es necesario obtener una referencia al objeto clase en sí.

Para remediar esto, podemos extender el cálculo multinivel de modo que el operador `new` enriquezca automáticamente a cada instancia con un atributo implícito `class` que contenga una referencia a la entidad que se está instanciando, si se desea acceder explícitamente a los atributos de la clase, o, si se prefiere que todos los atributos de una entidad permanezcan como atributos de sus instancias, es posible alterar la semántica de la instanciación de un atributo de modo que al instanciar un atributo de potencia cero el resultado sea un nuevo atributo de potencia cero, en vez de quedar eliminado en el proceso de instanciación.

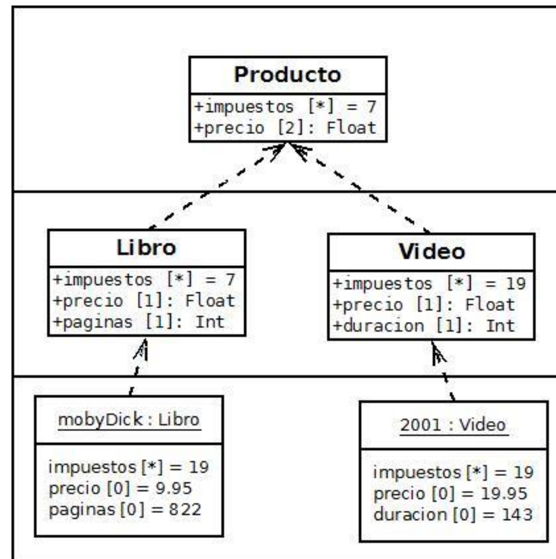
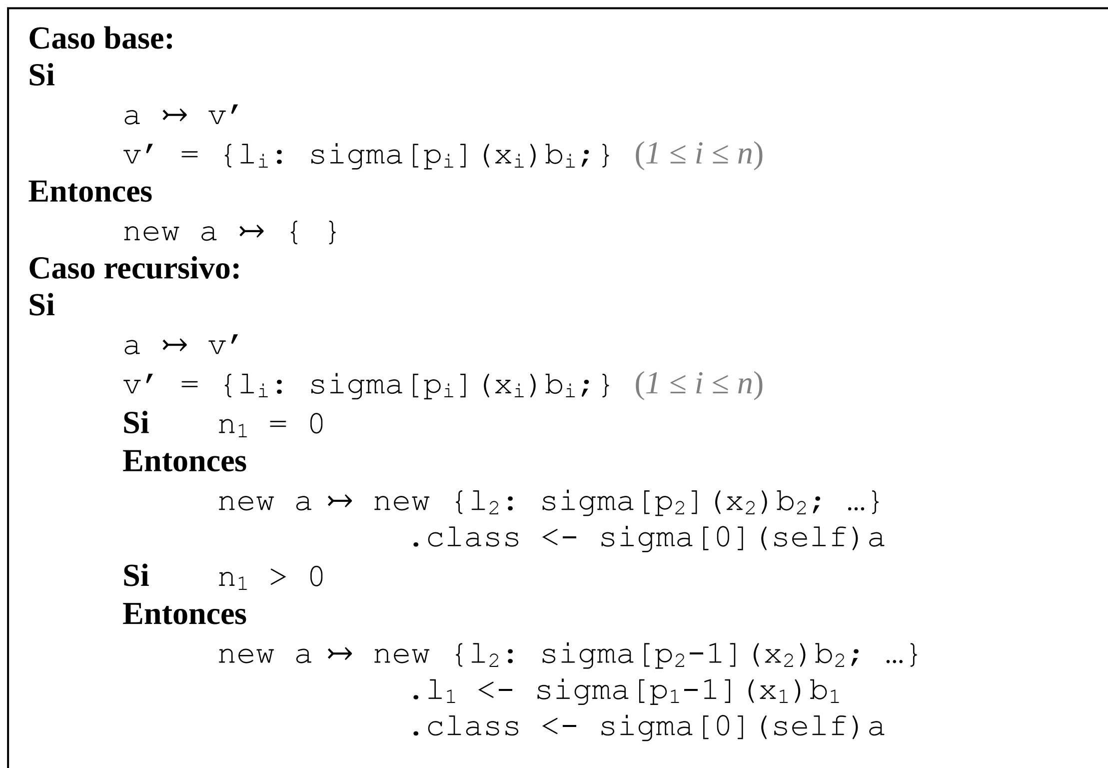


Figura 16. Atributo impuestos *presente en varios niveles del modelo*

Si se desea seguir el primer enfoque, por ejemplo, basta modificar la regla 4 del lenguaje de la siguiente manera.

- Regla 4b: instanciación de un objeto



Caso base:
Si

```
a ↣ v'
v' = {l_i: sigma[p_i](x_i)b_i;} (1 ≤ i ≤ n)
```

Entonces

```
new a ↣ { }
```

Caso recursivo:
Si

```
a ↣ v'
v' = {l_i: sigma[p_i](x_i)b_i;} (1 ≤ i ≤ n)
```

Si $n_1 = 0$
Entonces

```
new a ↣ new {l_2: sigma[p_2](x_2)b_2; …}
            .class <- sigma[0](self)a
```

Si $n_1 > 0$
Entonces

```
new a ↣ new {l_2: sigma[p_2-1](x_2)b_2; …}
            .l_1 <- sigma[p_1-1](x_1)b_1
            .class <- sigma[0](self)a
```

Al introducir esta modificación, todo objeto del lenguaje sigma creado con la sentencia `new` pasa a tener un puntero a su clase como parte de su estructura. Si se implementa esta opción, parece lógico prohibir la etiqueta `class` como nombre de atributo, ya que colisionaría con la definida implícitamente, o usar un

símbolo en vez de la palabra `class` para denotar el acceso a la clase de un objeto. En el *Listado 5* se puede ver un ejemplo de cómo usar el cálculo alfa extendido con la regla 4b para implementar el modelo multinivel propuesto en la *Figura 4*. Este modelo supone una pequeña extensión del cálculo alfa en el que se han implementado constantes enteras y en coma flotante, así como operaciones entre las mismas. Esta clase de extensiones se pueden introducir trivialmente en el lenguaje. Nótese el uso de un objeto para emular un espacio de declaraciones global.

```
{
      producto: sigma[0](global){
            impuestos[1](self): 10;
            precio[2](self): 0.0;
            precioFinal[2](self):
                  self.precio + self.class.impuestos;
      };

      libro: sigma[0](global)(new global.producto)
            .impuestos <- sigma[0](self)7
            .paginas   <- sigma[1](self)0;

      video: sigma[0](global)(new global.producto)
            .impuestos <- sigma[0](self)19
            .duracion  <- sigma[1](self)0;

      moby_dick: sigma[0](global)(new global.libro)
            .precio  <- sigma[0](self)9.95;
            .paginas <- sigma[0](self)822;

      el_padrino: sigma[0](global)(new global.video)
            .precio   <- sigma[0](self)19.95
            .duracion <- sigma[0](self)143;
}
```

Listado 5. Modelado multinivel en alfa

4.3. Beta: añadiendo tipos al cálculo multinivel

El cálculo alfa es razonablemente similar al cálculo sigma, con la excepción de la introducción de la potencia. Sin embargo, al intentar introducir un sistema de tipos, la situación se vuelve drásticamente más compleja. Esto se debe a que la interacción que tiene lugar en un lenguaje multinivel entre tipos y objetos es mucho más rica y dinámica que en un lenguaje convencional. Al adquirir los tipos una faceta de objetos será posible construir expresiones cuyo tipo se obtiene de la ejecución de otra expresión, por lo que la comprobación de tipos en el lenguaje se complica. En contraste, los tipos en el cálculo sigma de primer orden expuesto en 3.3 eran sintáctica y semánticamente más simples que los objetos.

Para entender hasta qué punto esto puede llegar a ser un problema, considérese una cierta expresión ‘`e`’ que represente un objeto en tiempo de ejecución, es decir, uno de los datos que maneja el programa.

Idealmente, dado un tipo '`t`' queremos poder comprobar si la expresión `e` tiene tipo `t` o no, es decir, si `e : t`. Sin embargo, y en contraste con sistemas de tipos más simples, el tipo `t` también tiene una faceta de objeto, es decir, consiste en una expresión computacional que reside en el nivel inmediatamente superior a `e`. La naturaleza de la expresión `t` no está, en principio, sujeta a restricciones: puede ser un tipo literal, una llamada a un método o una expresión más elaborada. Para verificar `e : t` se hace, por tanto, necesario ejecutar la expresión `t`. Ante este problema, se nos presentan tres soluciones alternativas: la primera posibilidad, empleada por deepJava (7), es delegar algunas de las comprobaciones de tipos hasta la ejecución del programa. Bajo este enfoque, un tipo puede ser el resultado de una computación arbitrariamente compleja, pero la detección de errores de tipos se delegaría hasta la ejecución del programa en sí. Esto elimina buena parte de la utilidad de un sistema de tipos, del que se espera que sirva para garantizar ciertos invariantes de un programa, por lo que hemos decidido no emplear esta solución.

Una segunda solución es permitir, como en la solución anterior, a computaciones arbitrarias tomar el papel de tipos, pero ejecutar las mismas en tiempo de compilación. Este enfoque es apenas más complejo que el anterior, pero tiene un grave inconveniente: obliga al compilador a ejecutar una porción del programa que está compilando, lo cual puede llevar a elevados retrasos o incluso bucles infinitos compilando un programa. Este comportamiento, en la práctica, no resulta excesivamente negativo, e incluso C++ (43), uno de los lenguajes de programación más populares hoy en día, adolece del mismo debido al funcionamiento de las plantillas (44).

La que consideramos la solución más satisfactoria consiste en limitar de algún modo la capacidad expresiva que se permite en expresiones usadas como tipos, de modo que el compilador pueda ejecutarlas sin entrar en un bucle infinito. Consideramos que la consecuente reducción en expresividad que esto causa al lenguaje es admisible, ya que se considera la norma en la gran mayoría de lenguajes de programación modernos (sólo unos pocos, como C++, permiten ejecutar computaciones arbitrarias a nivel de tipos). Para implementar esta solución, vamos a desarrollar un cálculo formal inicial incapaz de expresar computaciones cuya reducción nunca termine y, a continuación, añadiremos una primitiva a este lenguaje permitiendo bucles arbitrarios de manera controlada, es decir, en los objetos del nivel base, que consideramos que corresponderá a las entidades que existan durante la ejecución del programa. Así, aunque se introduce una cantidad arbitraria de niveles al lenguaje, efectivamente estamos estableciendo una forma más débil de la división entre valores y tipos presente en lenguajes que usan un modelo clásico de dos capas.

Una consideración adicional es el tratamiento de la potencia de una entidad (objeto/tipo). El modelado multinivel permite a una entidad tener asociado tanto un nivel como una potencia, lo que permite modelar conceptos como clases abstractas. Sin embargo, por simplicidad, vamos a considerar que la

potencia de todo objeto corresponde siempre a su nivel, tomando el concepto de método abstracto como primitivo.

m ::=	Cuerpo de un método	
a		Expresión (método concreto)
abstract		Método abstracto
a, b, c ::=	Expresiones	
x, y, z...		Variables
[**nivel**]{sigma[$\mathbf{p_i}$]($\mathbf{x_i}$)$\mathbf{m_i}$:$\mathbf{B_i}$;}	*(1 ≤ i ≤ n)*	Objeto literal
a.l		Selección de atributo
a.l <- sigma(**x**)**b**		Actualización de atributo
(**a**)		Expresión entre paréntesis

Tabla 6. Sintaxis de beta

Las principales diferencias con la sintaxis del cálculo sigma de primer orden son la introducción de indicadores del nivel de un objeto y la potencia de sus atributos, la posibilidad de declarar un método como abstracto anotando su tipo y, notablemente, la ausencia de una clase sintáctica de tipos: los tipos y las expresiones son fundamentalmente idénticos. La semántica de este sistema es igual en todos los sentidos a la semántica del cálculo alfa introducido en la sección anterior, introduciendo las modificaciones necesarias para acomodar las particularidades sintácticas del cálculo beta, y teniendo en cuenta las siguientes observaciones:

- Al instanciar un cierto objeto de nivel n, el nivel del objeto resultante tendrá nivel n – 1.
- Instanciar un objeto de nivel 0, invocar un método abstracto o sobreescribir un método no abstracto de un objeto no tienen una semántica definida: la expresión resultante queda trabada.

Las reglas de tipado de beta, aunque son paralelas a las reglas del cálculo sigma de primer orden, difieren de las mismas en maneras sutiles. Una diferencia principal es que, al ser los tipos también expresiones, es necesario definir una noción más relajada de la igualdad entre dos tipos: pretendemos considerar a dos tipos iguales si al ejecutarlos (reducirlos), tras un cierto número de pasos dan el mismo resultado. Para reflejar esto, vamos a introducir una nueva noción de igualdad entre tipos, que denotaremos con el símbolo ‘==’, que denota que dos tipos son iguales al ejecutarse.

- Igualdad de tipos

```
Si
      A = B
Entonces
      A == B
Si
      A ↣ A′
      B ↣ B′
      A′ == B′
Entonces
      A = B
```

Esta regla implica que para comprobar si dos tipos son iguales es necesario reducir los mismos a una forma común. Para garantizar que esta reducción termina y, por tanto, la comprobación puede llevarse a cabo en tiempo de compilación, es necesario que todas las computaciones que se puedan llevar a cabo con tipos en el cálculo beta terminen.

- Tipado bajo igualdad

```
Si
      E ⊢ a : A
      A == A′
Entonces
      E ⊢ a : A′
```

Esta regla es necesaria para transmitir la noción de igualdad definida con la regla anterior al sistema de tipos.

Una vez definida esta noción de igualdad y su impacto en el tipado, podemos pasar a definir el resto de reglas de tipado del cálculo beta, que resultan muy similares a las del cálculo sigma no tipado.

- Regla 1: objetos literales

```
Si
Entonces
      E ⊢ [j]{} : [j+1]{}
Si
      E ⊢ [j]{li: sigma[pi](xi)mi:Bi;}
            : [j+1]{li: sigma[pi+1](xi)mi:Bi;}
      E, xn+1 : [j+1-pn+1]{li: sigma[pi+1-pn+1](xi)mj:Bj;}
            ⊢ mn+1 : Bn+1
      mn+1 ≠ abstract o pn+1 > 0
Entonces
      E ⊢ [j]{li: sigma[pi](xi)mi:Bi; …;
                 ln+1: sigma[pn+1](xn+1)mn+1:Bn+1;}
            : [j+1]{li: sigma[pi+1](xi)mi:Bi; …;
                       ln+1: sigma[pn+1+1](xn+1)mn+1:Bn+1;}
```

(Para simplificar la notación, queda implícito en esta regla que el cuerpo de todo método abstracto es exactamente del tipo que se ha declarado, y que los métodos de potencia negativa quedan borrados.)

Esta regla, planteada recursivamente, es muy distinta de su homóloga en el cálculo sigma de primer orden. Esto se debe a dos factores: primero, es necesario eliminar la circularidad, para impedir crear objetos cuyos métodos introduzcan una dependencia circular. De este modo, evitamos recursiones infinitas y garantizamos que la evaluación de toda expresión beta termina. Además, esta regla ha de contemplar la potencia de cada método, de modo que un método de potencia p acceda a los atributos de potencia p como si fueran de potencia 0, a los atributos de potencia p+1 como de potencia 1, etcétera.

- Regla 2: selección de un atributo

```
Si
      E ⊢ a : [j]{li: sigma[pi](x)mi:Bi;}
      1 ≤ k ≤ n
      pk = 1
Entonces
      E ⊢ x.lk : Bk
```

Esta regla es prácticamente idéntica a la correspondiente en el cálculo sigma de primer orden. La única diferencia es que se comprueba que la potencia de un atributo sea la adecuada (nótese que un atributo de potencia 1 en el tipo de una expresión corresponde a un atributo de potencia 0 en la expresión).

- Regla 3: actualización de un atributo

```
Si
      E ⊢ a : [j]{lᵢ: sigma[pᵢ](xᵢ)mᵢ:Bᵢ;}
      l = lₖ con 1 ≤ k ≤ n
      mₖ = abstract
      Aₖ = [j]{lᵢ: sigma[pᵢ](xᵢ)mᵢ:Bᵢ;}  (1 ≤ i < k)
      E, x : Aₖ ⊢ b : Bₖ
Entonces
      E ⊢ a.l <- sigma[pₖ](x)b:Bₖ : A
Si
      A = [j]{lᵢ: sigma[pᵢ](xᵢ)abstract:Bᵢ;}
      E ⊢ a : A
      l no está entre los lᵢ
      E, x : A ⊢ b : B
      p < j
Entonces
      E ⊢ a.l <- sigma[p](x)b:B
            : [j]{l₁:sigma…lₙ…; l:sigma[p+1](x)abstract:B}
```

A pesar de la aparente complejidad de esta regla, la única diferencia sustancial con respecto a la regla equivalente para el cálculo lambda de primer orden es que en este cálculo hemos decidido sólo permitir instanciar un campo abstracto o añadir un campo nuevo. Esto no altera la capacidad expresiva del lenguaje, y resulta importante para preservar la consistencia del sistema. Para ilustrar esto, supongamos que se permite modificar un método no abstracto ya existente, y consideremos una hipotética extensión de beta equipada con tipos primitivos `int` y `string`, que residen a nivel 1, y cuyo tipo a nivel 2 es el tipo primitivo `primitive`. El código del *Listado 6* ejemplifica la clase de problemas que cabe encontrar al permitir la redefinición arbitraria de métodos existentes en un objeto: al redefinir el método `type`, el método `val`, cuyo tipo dependía del valor de `self.type`, pasa a tener un tipo incorrecto.

```
([1]{
      type: sigma[1](self)int:primitive;
      val:  sigma[1](self)1:self.type;
}).type <- sigma[1](self)string:prim
```

Listado 6. Incoherencia de tipos al sobreescribir métodos concretos.

Es posible pensar que este problema viene dado porque el *Listado 6* viola los preceptos del metamodelado estricto, ya que las instancias del objeto definido en el mismo serían entidades de nivel 0 y, sin embargo, contendrían una referencia a la entidad `int` de nivel 1. Al

restringirnos a pies juntillas al metamodelado estricto sospechamos que este problema desaparecería por completo. Sin embargo, esta restricción haría imposible expresar patrones comunes como acceder a métodos estáticos de una clase desde una instancia de la misma. Si se soluciona esta restricción, el problema presente en el *Listado 6* se vuelve a hacer patente.

- Regla 4: instanciación de una entidad

Si
$E \vdash$ `a : [j]{`l_i`: sigma[`p_i`](x)`m_i`:`B_i`;}`
$j > 1$
Para todo i con m_i = abstract se tiene $p_i > 2$
Entonces
$E \vdash$ `new a : [j-1]{`l_i`: sigma[`p_i`-1](x)`m_i`:`B_i`;}`

De nuevo, para simplificar la notación, hemos asumido implícitamente que los métodos cuya potencia pase a ser negativa quedan eliminados del tipo de la expresión resultante.

Además de las reglas de tipado, hemos escogido unas reglas de subtipado, que incluyen las reglas de subtipado 1, 2 y 4 del cálculo sigma de primer orden (estas reglas de subtipado están presentes en una forma u otra en todos los sistemas de tipos que admiten esa noción). Hay que tener en cuenta que la relación de subtipado en el cálculo beta se establece entre dos expresiones y no, como era el caso en el cálculo sigma de primer orden, entre una expresión y un tipo, al no haber una distinción entre estas dos categorías. Adicionalmente a las reglas 1, 2 y 4, introduciremos la siguiente modificación de la regla 3:

- Regla 3b: tipos de objetos literales

Si
`a = [nivel]{ `l_i`: sigma[`p_i`](`x_i`)`m_i`:`a_i`; }` $(1 \leq i \leq n)$
`b = [nivel]{ `s_j`: sigma[`q_j`](`y_j`)`k_j`:`b_j`; }` $(1 \leq j \leq m)$
Para todo j entre 1 y m, s_j corresponde a uno de los l_i, l_{ij}.
$b_j == a_{ij}$
$q_j = p_{ij}$
Entonces
`a <: b`

Aunque este lenguaje no permite expresar computaciones arbitrarias por no permitir recursión, se pueden expresar muchos patrones básicos. En el *Listado 7* podemos ver una traducción tipada del código del *Listado 5* a beta. Nótese la excesiva redundancia en las anotaciones de tipos, que obliga a duplicar mucha información. Es posible implementar algoritmos simples de inferencia de tipos que permitan omitir algunas

de estas anotaciones e inferirlas del contexto. Otra particularidad del sistema es que las entidades del nivel más superior del modelo, `producto` en el ejemplo, necesitan una anotación de tipo explícita, completa; mientras que las entidades de niveles inferiores solamente requieren anotaciones de las extensiones lingüísticas que introducen, como los atributos `precio` o `duración` en el ejemplo.

Estas reglas de tipado y subtipado, tal y como quedan recogidas aquí, describen un lenguaje simple que no se acoge a las restricciones del metamodelado estricto, ya que en ningún momento se ha incluido ninguna condición sobre la relación entre el nivel de un elemento y el nivel de sus atributos. Es sencillo incluir estas restricciones si se desea estudiar un sistema de tipos basado en el metamodelado estricto: basta añadir a las reglas de tipado de un objeto literal una cláusula que verifique que el nivel de los tipos de cada método se ajusta al nivel adecuado, teniendo en cuenta el nivel del objeto literal y la potencia de cada método, resultando en un cálculo formal al que podemos denominar beta-e.

```
[0]{
      producto: sigma[0](global)[2]{
            impuestos:   sigma[1](self)abstract:int;
            precio:      sigma[2](self)abstract:double;
            precioFinal: sigma[2](self) self.precio + self.class.impuestos
                  :double;
      }:sigma[0](global)[3]{
            impuestos:   [2](self)abstract:int;
            precio:      [3](self)abstract:double;
            precioFinal: [3](self)abstract:double;
      };

      libro: sigma[0](global)new (global.producto
            .impuestos <- sigma[1](self)7:int
            .paginas   <- sigma[2](self)abstract:int)
            :global.producto.paginas <- sigma[2](self)abstract:int;

      video: sigma[0](global)new (global.producto
            .impuestos <- sigma[1](self)19:int
            .duracion  <- sigma[2](self)abstract:int)
            :global.producto.duracion <- sigma[2](self)abstract:int

      moby_dick: sigma[0](global)new (global.libro
            .precio  <- sigma[1](self)9.95:double
            .paginas <- sigma[1](self)822:int)
            :global.libro;

      el_padrino: sigma[0](global)new (global.video
            .precio   <- sigma[1](self)19.95:double
            .duracion <- sigma[1](self)143:int)
            :global.video;
}
```

Listado 7. Modelado multinivel en beta

Sin embargo, cuando se contempla la extensión del lenguaje con nuevas primitivas, el modelado estricto introduce un nuevo problema: suponiendo que extendemos el lenguaje básico beta-e con una

primitiva `int : primitive` a nivel 1. Supongamos además un programa como el ejemplificado en el *Listado 8*. En términos de un lenguaje más convencional, el mismo define una clase abstracta producto, y una subclase de la misma denominada libro, con un método estático que devuelve un número entero, tal y como se ve en el *Listado 8*. Sin embargo, en esta extensión del lenguaje, el número 10 es una entidad de nivel 0 y, si nos ceñimos a las restricciones del metamodelado estricto, no es posible referenciarla desde una entidad de nivel 1, tal y como se muestra en la *Figura 17*.

```
[1]{
      producto: sigma[0](self)[1]{
            impuestos: sigma[0](self)abstract:int;
      }:[2]{impuestos: sigma[1](self)abstract:int};

      libro: sigma[0](self)
            (self.producto.impuestos
                  <- sigma[0](self)10:int)
            :[2]{impuestos: sigma[1](self)abstract:int};
}
```

Listado 8. Violación del metamodelado estricto

Si se desean mantener las restricciones del metamodelado estricto pero permitir código como el *Listado 8* se pueden aplicar varias soluciones: se puede extender el lenguaje beta con potencias indefinidas, de modo que la entidad int pueda realizarse en cualquier nivel del modelo, o se pueden establecer reglas especiales para los tipos primitivos (esta solución es la implementada en deepJava (7)). Una solución alternativa es considerar que los valores primitivos únicamente tienen un tipo lingüístico, y no un tipo ontológico, por lo que no están sujetos a las restricciones del metamodelado estricto.

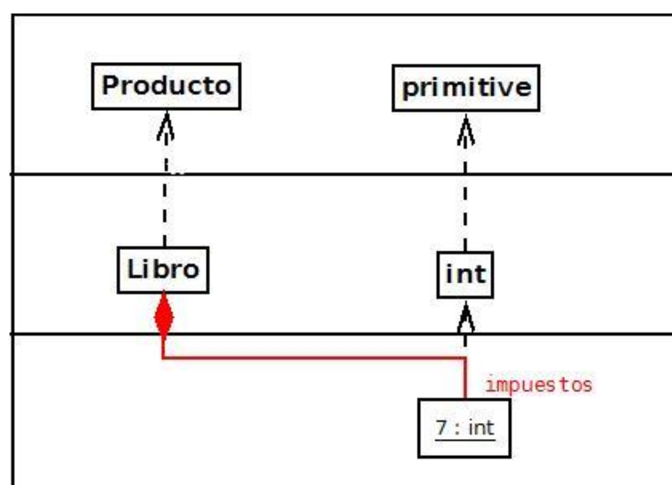


Figura 17. Estructura de clases del Listado 8

5. Desarrollo de compiladores

5.1. Motivación

Los cálculos formales definidos en la sección anterior constituyen un primer paso fundamental en la definición rigurosa de lenguajes de programación multinivel. Sin embargo, con objeto de probar la aplicabilidad práctica de estos cálculos, se ha considerado relevante el desarrollo de compiladores basados en sendos cálculos formales. Esto ha proporcionado una perspectiva muy valiosa al desarrollo del sistema de tipos del cálculo beta, ya que las primeras versiones del mismo, desarrolladas sin una implementación en mente, tenían la grave desventaja de que los algoritmos de comprobación de tipos resultantes no hubieran sido computables, al permitir bucles y recursión arbitraria a cualquier nivel.

Además de su utilidad en la definición de los cálculos formales mostrados en el apartado 4, estos lenguajes tienen utilidad práctica, y pueden emplearse ya para la definición de modelos ejecutables en JavaScript. Esta noción de modelos ejecutables resulta interesante ya que aunque hasta el momento el uso de modelos en el desarrollo de software se limita a la fase de diseño, el uso de modelos en tiempo de ejecución se ha propuesto como herramienta para desarrollar sistemas más adaptables y flexibles (45).

Es importante señalar que los lenguajes y compiladores presentados aquí son altamente experimentales y, por el momento, no están listos para ser empleados para el desarrollo de sistemas reales, aunque esperamos que lleguen a estarlo en el futuro. Cabe esperar cambios sustanciales en el código y funcionamiento de ambos compiladores, así como en la sintaxis de los lenguajes definidos aquí. El código de ambos compiladores se está desarrollando como código abierto y está disponible actualmente en la plataforma GitHub. El *Anexo I* contiene instrucciones detalladas para obtener, compilar y usar los compiladores.

5.2. Tecnología empleada

Para implementar los dos compiladores realizados hemos empleado el lenguaje de programación OCaml (46). Este lenguaje presenta algunas ventajas respecto a alternativas más ortodoxas, como Java o C/C++, especialmente en el área del desarrollo de compiladores: por un lado, su empleo de tipos variantes permite fácilmente representar estructuras como árboles de sintaxis y transformaciones recursivas sobre las mismas. Adicionalmente, la implementación estándar de OCaml incluye las herramientas ocamllex y

ocamlyacc (47) para la generación de analizadores léxicos y sintácticos, similares a flex y bison, que han simplificado enormemente el proceso de desarrollo.

Además de estas ventajas, OCaml es un lenguaje de programación funcional, especialmente apto para manejar estructuras recursivas, presentes en el desarrollo de lenguajes de programación en forma de árboles de sintaxis. Sin embargo, al tener un sistema de tipos estático y compilar a código nativo, se puede mantener un nivel de eficiencia considerable (48).

Inicialmente se consideró emplear el sistema K (49), una plataforma basada en Maude (50) especializada en el desarrollo y análisis de lenguajes de programación, que se ha empleado exitosamente para formalizar la semántica de lenguajes como Java o C (51; 52; 53). Sin embargo, la misma tiene una curva de aprendizaje considerable, al estar basada en un modelo de reducción simultánea inspirado por la Máquina Abstracta Química (54) poco familiar, y que resulta en una reducción considerable de la eficiencia, tanto del compilador de K, que genera intérpretes de un lenguaje a partir de una definición del mismo, como de los intérpretes resultantes. Por esto, a la larga resultó más efectivo migrar el código K existente a OCaml.

Como objetivo de compilación hemos escogido el lenguaje de programación JavaScript. Este lenguaje tiene una serie de características que lo hacen preferible para este proyecto a código nativo, LLVM o una máquina virtual propia: se trata de un lenguaje de alto nivel, en el que se pueden expresar fácilmente estructuras asociativas como objetos, y que permite emplear funciones para manejar implícitamente el contexto léxico de una expresión, evitando tener que emplear la pila para mantener un registro de variables locales. A pesar de ser un lenguaje dinámico, proporciona un rendimiento relativamente considerable comparado con otros lenguajes como Ruby o Python, y a diferencia de este último su sintaxis es flexible, lo cual permite generar código JavaScript con facilidad.

5.3. El lenguaje Arche y su compilador

El lenguaje Arche[1] es un lenguaje muy simple basado en el cálculo alfa. Este es un lenguaje muy simple que tiene como objetivo ilustrar una posible manera de compilar código multinivel a código orientado a objetos. Por ello, la semántica de Arche es fundamentalmente idéntica a la del cálculo alfa, y sólo se diferencia del mismo en aspectos superficiales.

Como cabe esperar, los objetos en Arche quedan representados por objetos en JavaScript. Sin embargo, los métodos de un objeto no se compilan directamente a funciones JavaScript, sino a arrays cuyo

[1] Del griego antiguo ἀρχή, “origen”, en la filosofía griega el elemento inicial del que provienen todas las entidades que existen.

primer elemento es la potencia del método y su segundo elemento es el cuerpo del método en sí, representado como una función JavaScript. Debido a esta representación, la invocación de un método en Arche no se puede compilar directamente a una invocación de método en JavaScript, sino que es necesario extraer la potencia del método, comprobar su valor y proceder a ejecutar el cuerpo de dicho método, ligándolo al objeto sobre el que se invoca. Para facilitar este procedimiento, en vez de generar código para ejecutar el mismo en cada llamada a método se emplea la primitiva invoke, cuya definición es añadida por el compilador al código de todo programa Arche.

La instanciación de un objeto Arche consiste, a nivel de JavaScript, en crear un nuevo objeto y copiar los métodos del objeto que se está instanciando, decrementando la potencia de cada uno y eliminando los métodos de potencia cero. Este proceso se ha implementado como una función escrita en JavaScript nativo, la función instantiate, cuya definición se añade automáticamente al programa durante el proceso de compilación.

Para facilitar la extensión de objetos, el lenguaje Arche introduce la estructura sintáctica '`extend` $\mathbf{e_1}$ `with` $\mathbf{e_2}$', que sustituye a la actualización de un campo en el cálculo alfa. Esta estructura genera una copia del objeto denotado por la expresión 'e_1', al que se le añaden los métodos del objeto denotado por 'e_2'. Esto permite extender un objeto con varios campos a la vez, y da más flexibilidad que la actualización de un único campo existente en el cálculo alfa. Para ejecutar esta extensión se ha implementado una tercera función estándar, extend, que también es añadida por el compilador.

Otrs extensión que Arche proporciona frente al cálculo es la creación de funciones: el lenguaje Arche, además de objetos, permite crear funciones empleando la sintaxis '`function (`**`args`**`…) ->` **`expr`** `end`'. Esto no proporciona ninguna ganancia expresiva al lenguaje, ya que es posible emular funciones en el cálculo alfa puro, pero esta extensión constituye una conveniencia sintáctica considerable y, al compilarse directamente a una función JavaScript, proporciona un mejor rendimiento.

A nivel de implementación, el compilador de Arche es muy poco sofisticado y no realiza niguna clase de optimización: los programas en Arche se compilan a JavaScript de manera directa, pasando por una sencilla representación intermedia. El código es muy simple ya que al adoptar JavaScript como objetivo no es necesario tener en cuenta detalles como asignación de registros, implementación de tablas de símbolos o la representación a bajo nivel de los objetos. El compilador está pues estructurado en cinco módulos:

- Ast (ast.ml): el módulo Ast contiene la declaración de los tipos que componen el árbol de sintaxis del lenguaje Arche.
- Lexer (lexer.ml, generado a partir de lexer.mll): el módulo Lexer, generado por ocamllex a

partir de las expresiones regulares contenidas en lexer.mll, contiene la definición de los lexemas empleados en la gramática de Arche.

- Parser (parser.ml, generado a partir de parser.mly): el módulo Parser, generado por ocamlyacc a partir de la gramática contenida en parser.mly, contiene métodos para parsear una cadena de caracteres a un árbol de sintaxis de Arche.
- Compiler (compiler.ml): el módulo Compiler exporta la función compile_program, que compila un programa en Arche dado como un AST y lo convierte en un árbol de sintaxis de JavaScript.
- Arche (arche.ml): el módulo Arche es el punto de entrada del compilador, que lee del fichero especificado (o la entrada estándar) un programa Arche y emite el resultado de su compilación a JavaScript.

El *Listado 9* muestra una traducción del código alfa en el *Listado 5* al lenguaje Arche, ejemplificando la sintaxis del mismo. En el *Anexo II* se puede encontrar el código ocamlyacc que define la sintaxis completa de Arche.

```
{
      producto := (global) {
            impuestos   [1] := 10;
            precio      [2] := 0;
            precioFinal [2] := (self) self.precio + self.class.impuestos;
      };

      libro := (global) extend new(global.producto) with {
            impuestos [0] := 7;
            paginas   [1] := 0;
      };

      video := (global) extend new(global.producto) with {
            impuestos [0] := 19;
            duracion  [1] := 0;
      };

      moby_dick := (global) extend new(global.libro) with {
            paginas := 822;
            precio  := 9;
      };

      el_padrino := (global) extend new(global.video) with {
            duracion := 143;
            precio   := 19;
      };
}
```

Listado 9. Implementación de un modelo multinivel en Arche

5.4. El lenguaje Archetype y su compilador

Con el objetivo de motivar el cálculo beta y ayudar a la determinación de un conjunto de reglas de tipado para las que sea posible establecer un algoritmo computable, hemos desarrollado el lenguaje Archetype, un lenguaje multinivel orientado a objetos con tipado estático que genera JavaScript a través de la infraestructura de Arche.

El lenguaje Archetype no incluye la totalidad de las características presentes en el cálculo beta, con el objetivo de mantener la implementación lo más simple posible. Debido a esto se ha escogido la gramática con el objetivo de que resulte lo más simple posible, ya que el generador de parsers ocamlyacc tiene una capacidad limitada de resolver conflictos (47). Aparte de estas diferencias sintácticas menores, la limitación más inmediata de Archetype respecto del cálculo beta es que, a diferencia de este último, Archetype no permite elegir explícitamente el nombre del objeto "self", sino que el mismo es siempre es el identificador `self`. Esta limitación se debe únicamente al deseo de simplificar al máximo la implementación del compilador, pero esperamos eliminarla por completo en el futuro. A pesar de esta limitación importante, Archetype ofrece una serie de extensiones útiles al cálculo beta.

La más importante de estas extensiones es un sistema primitivo de inferencia de tipos: como se puede ver en el *Listado 7*, el cálculo beta es muy redundante debido a la necesidad de especificar explícitamente el tipo de cada método. En muchos casos esto es innecesario y el lenguaje Archetype permite omitir las anotaciones de tipos en métodos no abstractos, aunque siempre es posible anotar el tipo de un método explícitamente, si así se desea.

Además, este lenguaje proporciona una serie de extensiones de conveniencia, puramente sintácticas, para definir e instanciar objetos: por un lado, se añade al lenguaje la sintaxis '`inherit expr;`', que puede ocupar el lugar de un método en la declaración de un objeto literal y denota que el mismo extiende los métodos definidos por el objeto que resulta de evaluar expr. Del mismo modo, se incorpora al lenguaje un atajo sintáctico para instanciar tipos con métodos abstractos: la sintaxis '$\texttt{instantiate(e, } m_1[p_1] := e_1, \ldots, m_n[p_n] := e_n\texttt{)}$' permite instanciar directamente el objeto denotado por la expresión e, asignando los valores $e_1 \ldots e_n$ a sus métodos abstractos $e_1 \ldots e_n$. De este modo se consigue una sintaxis más similar a la que se emplea al invocar constructores de clases en un lenguaje orientado a objetos convencional. Un programa en Archetype permite también especificar una serie de declaraciones globales antes de escribir el cuerpo del programa. Esto evita la solución poco elegante de encerrar todo programa en un objeto que contenga variables globales, solución que en todo caso no resulta viable en la versión actual de Archetype, debido a la limitación sobre el nombre de la variable `self` mencionada anteriormente. Una última conveniencia es que en Archetype las declaraciones de la potencia de un método o del nivel de un

objeto son opcionales, asumiéndose cero su valor por defecto.

Un aspecto interesante de Archetype es que su estructura modular permite fácilmente incluir nuevas primitivas como extensiones del lenguaje. Para esto, basta modificar el parser para acomodar la sintaxis deseada y añadir una descripción de las entidades primitivas que se añaden al lenguaje al módulo Primitives. En esta, es necesario especificar las reglas de tipado, subtipado, evaluación y compilación de la entidad a definir. Para ilustrar la facilidad de introducir primitivas, el lenguaje Archetype incluye como primitivas números enteros y booleanos (definidos a nivel 0) y sus tipos int y bool respectivamente (definidos a nivel 1), así como un conjunto de operaciones elementales sobre los mismos.

La definición del parser de Archetype se puede encontrar íntegramente en el *Anexo III*. Un ejemplo de su uso se puede ver en el *Listado 10*, que muestra una versión tipada del programa Arche del *Listado 9*.

```
producto := object [2]{
      - impuestos [1] : int;
      - precio    [2] : int;
};

libro := object[1] {
      inherit instantiate(producto, impuestos := 7);
      - paginas [1] : int;
};

video := object[1] {
      inherit instantiate(producto, impuestos := 19);
      - duracion [1] : int;
};

begin
    object{
      - moby_dick  := instantiate(libro,
            paginas := 822,
            precio  := 9
      );
      - el_padrino := instantiate(video,
            duracion := 143,
            precio   := 19
      );
    }
end
```

Listado 10. Implementación de un modelo multinivel en Archetype

A nivel técnico, el compilador de Archetype es considerablemente más complejo que el compilador de Arche, ya que además de transformar el código Archetype en código JavaScript realiza una labor de comprobación de tipos adicional. El compilador de Archetype comprende los siguientes módulos:

- Ast (ast.ml): el módulo Ast contiene la declaración de los tipos que componen el árbol de

sintaxis del lenguaje Arche.

- Lexer (lexer.ml, generado a partir de lexer.mll): el módulo Lexer, generado por ocamllex a partir de las expresiones regulares contenidas en lexer.mll, contiene la definición de los lexemas empleados en la gramática de Archetype.
- Parser (parser.ml, generado a partir de parser.mly): el módulo Parser, generado por ocamlyacc a partir de la gramática contenida en parser.mly, contiene métodos para parsear una cadena de caracteres a un árbol de sintaxis de Archetype.
- Js (js.ml): el módulo Js proporciona una representación simplificada del árbol de sintaxis de JavaScript, y permite convertir el mismo a una cadena de texto con código JavaScript válido.
- Reporting (reporting.ml): el módulo Reporting define una serie de excepciones comunes al programa así como funciones auxiliares para mostrar errores.
- Compiler (compiler.ml): el módulo Compiler exporta las primitivas para convertir el AST de Archetype en un string que contiene el código JavaScript resultante de la compilación.
- Archetype (archetype.ml): el módulo Archetype es el punto de entrada del compilador. Se encarga de leer un programa Archetype de la entrada, convertirlo a su AST e invocar el compilador sobre el mismo.

6. Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo hemos logrado definir dos cálculos formales alfa y beta que recogen los aspectos fundamentales de la semántica del modelado multinivel. Para verificar su aplicabilidad práctica, hemos desarrollado sendos lenguajes de programación Arche y Archetype, basados en alfa y beta respectivamente. Este proceso no ha tenido una única dirección, sino que el trabajo en los compiladores de estos lenguajes ha permitido refinar y ajustar las reglas de los respectivos cálculos, especialmente en el caso de las reglas de tipado de beta, cuya tratabilidad por un compilador resulta fundamental.

Al revisar la literatura existente durante este proceso de desarrollo, tanto sobre modelado multinivel como sobre cálculos formales, y desarrollar una formalización de la programación multinivel, resulta evidente que se trata de un área de investigación en la que hay una gran cantidad de terreno sin explorar: muchas de las nociones primitivas del modelado multinivel admiten generalizaciones o varias definiciones distintas, que constituyen avenidas a explorar. Los modelos formales desarrollados aquí pueden ayudarnos a identificar estas nociones primitivas y estudiar su interacción entre sí y bajo distintas generalizaciones.

En una dimensión más práctica, parece evidente que los lenguajes clásicos, basados en una arquitectura de dos niveles, no son suficientes para expresar ciertos patrones, que requieren de una arquitectura basada en el metamodelado profundo. Esperamos que los lenguajes de programación que hemos desarrollado puedan constituir una base para el desarrollo de futuras téncicas y lenguajes de programación que permitan aprovechar estos nuevos patrones de programación a varios niveles sin incurrir en la impedancia cognitiva causada por emplear un paradigma de modelado multinivel incompatible con los lenguajes de programación habituales.

En resumen, la principal conclusión de este trabajo es que el modelado multinivel para el desarrollo de software es una disciplina verdaderamente novedosa que requiere aún de mucho esfuerzo por parte de investigadores y desarrolladores, que esperamos pueda apoyarse en el trabajo que se ha realizado aquí.

6.2. Trabajo futuro en cálculos multinivel

Una línea de investigación prometedora es la formalización y catalogación sistemática de las

distintas variantes de la semántica de la potencia en los cálculos multinivel definidos aquí. Como se ha indicado, la noción de potencia admite varias generalizaciones distintas, y sería conveniente disponer de estudios comparativos sobre las mismas que especifiquen de manera precisa su semántica e investiguen su impacto en los distintos cálculos formales y su interacción con el resto de características de los cálculos multinivel.

Por otro lado, aunque los cálculos multinivel definidos aquí son apropiados como formalización de lenguajes de programación basados en metamodelado multinivel, es necesario un análisis más exhaustivo de los mismos: existen muchas propiedades deseables en un cálculo formal, como confluencia o normalización fuerte (32). Estas son propiedades matemáticas cuya demostración constituye un esfuerzo considerable y no se ha emprendido en este trabajo, pero resulta un tema muy relevante de cara a trabajos futuros. Especialmente importantes resultarían las propiedades sobre el sistema de tipos del cálculo beta y una demostración matemática de su consistencia, es decir, de que en ningún momento las reglas de tipado definidas aquí asignan un tipo erróneo a una expresión.

Además, al desarrollar estos cálculos formales, se han realizado una serie de suposiciones y decisiones. Por ejemplo, identificar el nivel de una entidad con la potencia de la misma en el cálculo beta, tomar como primitiva la noción de método abstracto o el uso del tipado estructural. Parece pertinente estudiar el impacto de estas decisiones en la semántica de los cálculos multinivel, encontrando nuevas formulaciones de los mismos en términos de otras primitivas y estudiando su relación con el cálculo original para determinar qué comportamientos se pueden modelar en base a qué primitivas. Por ejemplo, el tipado clásico o nominal se puede emular en un sistema con tipado estructural, pero no al revés.

El cálculo beta es especialmente interesante por la libertad que ofrece su estudio. Como se ha explicado en la sección 4.3, para facilitar el proceso de comprobación de tipos se ha definido un lenguaje en el que la ejecución de un programa termina siempre. Esta restricción se puede relajar, delegando parte de la comprobación de tipos a la ejecución del programa. Un posible tema de investigación sería la definición de un sistema de tipos híbrido (55), de tal modo que existieran expresiones a las que no se pudiera asignar un tipo en tiempo de compilación a cambio de una mayor flexibilidad.

6.3. Trabajo futuro en Arche/Archetype

Los lenguajes Arche y Archetype están en una fase experimental, y es necesaria gran cantidad de trabajo para convertirlos en lenguajes prácticos. Una tarea vital es el desarrollo de bibliotecas estándar y tipos de datos primitivos para dichos lenguajes y el desarrollo de una máquina virtual adecuada para la ejecución de los mismos (o bien un compilador capaz de emitir código nativo). Además, resultaría

conveniente disponer de herramientas para trabajar con los mismos: soporte para editores, mecanismos de refactorización automática, etc.

Los compiladores en sí requieren aún de una gran cantidad de esfuerzo para optimizar el código que se produce. La implementación actual es simple y directa, pero resulta en un código muy ineficiente en tiempo de ejecución, por lo que es necesario mejorar los compiladores introduciendo fases de optimización en los mismos. Sería conveniente, además, disponer de baterías de pruebas para verificar el correcto funcionamiento de los mismos, una vez lleguen a un estado estable a partir del que no quepa esperar cambios radicales en estos lenguajes.

6.4. Aplicaciones potenciales

El uso de lenguajes de programación que generalizen la dualidad objeto/clase a una cantidad arbitraria de tipos es, hoy en día, nulo. Por tanto, no existe apenas información sobre los patrones de programación particulares que puedan emerger de estos lenguajes. Kühne (7) señala que el uso de lenguajes multinivel generaliza la noción de tipos genéricos, y se pueden emplear entidades de nivel 2 para obtener una codificación del polimorfismo acotado (30). Estas entidades, en cierto modo tipos de tipos, resultan similares a la idea de conceptos propuesta para estandarización en C++ (56), y a la noción de kinds (30) existente en teoría de tipos y realizada en el lenguaje de programación Haskell (57), lo cual parece indicar que ciertos conceptos avanzados en lenguajes de programación clásicos pueden expresarse de manera simple y elegante empleando un lenguaje multinivel.

Por otro lado, la combinación de tipado estructural e inferencia de tipos empleada en el lenguaje Archtype puede capacitar una poderosa forma de modelado ascendente o *bottom-up* (58; 59), ya que permite extraer jerarquías de tipos a partir de conjuntos particulares de instancias. Especialmente interesante resultaría la posibilidad del desarrollo de un 'anti-compilador' de estos lenguajes, que tomase como entrada un conjunto de objetos de nivel 0 y permitiese inferir la jerarquía de tipos a partir de la cual se han generado estos objetos.

Los sistemas de tipos multinivel parecen, además, especialmente indicados para formalizar y tipar sistemas de metaprogramación existentes en muchos lenguajes dinámicos (20; 25; 28) que emplean de manera natural una arquitectura en varias capas. Estos sistemas, aunque no tienen cabida en un lenguaje con tipado estático al violar la separación estricta entre tipos y valores, tal vez podrían ser expresados en un lenguaje tipado multinivel, permitiendo el uso de estas capacidades de metaprogramación de manera segura y eficiente.

Por último, es posible que ciertos conceptos del metamodelado multinivel tengan aplicaciones a la programación en etapas múltiples (60), un paradigma que estructura la compilación de un programa como una serie de fases intermedias, en cada una de las cuales se ejecuta un cierto programa para generar el código a ser ejecutado en la siguiente fase. Esta distribución del código en una jerarquía de fases, en la que las entidades de una fase de compilación condicionan la estructura de las entidades de fases subsiguientes parece presentar claros paralelos con las ideas centrales de la programación multinivel, en la que los tipos se organizan en niveles que determinan la estructura de las entidades de niveles inferiores mediante potencia.

7. Bibliografía

1. **Grady Booch, James Rumbaugh y Ivar Jacobson.** *The Unified Modeling Language User Guide (2nd Ed).* s.l. : Adison-Wesley, 2005. 978-0321267979.

2. **Juan de Lara, Esther Guerra, Ruth Cobos y Jaime Moreno-Llorena.** Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, 2013.

3. **OMG.** OMG's MetaObject Facility. [En línea] http://www.omg.org/mof/.

4. **Dave Steinberg et al.** *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008.

5. **Alessandro Rosini, Juan de Lara, Esther Guerra, Adrian Rutle, Uwe Wolter.** A Formalisation of Deep Metamodelling. *Formal Aspects of Computing.* 2014.

6. **Colin Atkinson y Thomas Kühne.** The Essence of Multilevel Metamodeling. *International Conference on The Unified Modeling Language, Modeling Languages, Concepts and Tools.* págs. 19-33. Springer, 2001.

7. **Thomas Kühne y Daniel Schreiber.** Can Programming be Liberated from the Two-Level Style? Multi-Level Programming with DeepJava. *Proceedings of OOPSLA 2007*, págs 229-244. ACM, 2007.

8. **Juan de Lara y Esther Guerra.** Deep Meta-Modelling with MetaDepth. *Proceedings of TOOLS 2010*, pags 1-20. Springer, 2010.

9. **Martin Abadi y Luca Cardelli.** *A Theory of Objects.* Springer, 1998.

10. **Colin Atkinson y Thomas Kühne**. Profiles in a Strict Metamodeling Framework. *Science of Computer Programming*, 44(1):5-22, 2002.

11. **Object Management Group.** *OMG Unified Modeling Language, Superstructure Specification, V2.1.2.* http://doc.omg.org/formal/2007-11-02.pdf.

12. **Colin Atkinson y Thomas Kühne.** Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 2003.

13. **Ralf Gitzel y Tobias Hildenbrand.** A Taxonomy of Metamodel Hierarchies. 2005.

14. **James Martin y James Odell.** *Object-Oriented Methods: A Foundation.* Prentice Hall, 1997.

15. **Cesar Gonzalez-Perez y Brian Henderson-Sellers.** A powertype-based metamodelling framework. *Software & System Modeling*, 5(1):72-90, 2006.

16. **Cesar Gonzalez-Perez y Brian Henderson-Sellers.** *Metamodelling for Software Engineering.*

Wiley, 2008.

17. **Colin Atkinson y Thomas Kühne**. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290-321, 2002.

18. Melanee: The Deep-modeling Domain-specific Language Workbench. http://www.melanee.org.

19. Dynamic type creation and names for built-in types. *Python 3.4.1 documentation*. https://docs.python.org/3/library/types.html.

20. Class: Class. *Ruby-Doc.org: Documenting the Ruby Language*. http://www.ruby-doc.org/core-2.1.2/Class.html.

21. **Nathaniel Nystrom, Michael Clarkson, Andrew Myers.** Polyglot: An Extensible Compiler Framework for Java. *Proceedings of the 12th International Conference on Compiler Construction*, págs. 138-152. Springer, 2003.

22. **Shigeru Chiba.** Load-time structural reflection in Java. *Proceedings of the 14th European Conference on Object-Oriented Programming*, págs. 313-336. Springer, 2000.

23. **Dimitrios Kolovos, Richard Paige y Fiona Polack.** The Epsilon Object Language. *Proceedings of the 2nd ECMDA-FA.* Springer, 2006.

24. **Martin Fowler.** *Domain Specific Languages.* Addison-Wesley.

25. Data Model. *Python 3.4.1 Documentation.* https://docs.python.org/3/reference/datamodel.html.

26. ABC - Abstract Base Classes. *Python 3.4.1 Documentation* https://docs.python.org/3/library/abc.html.

27. enum - Support for enumerations. *Python 3.4.1 Documentation.* https://docs.python.org/3/library/enum.html

28. **Adele Goldberg y David Robson.** *Smalltalk-80: The Language*. Addison-Wesley, 1983.

29. **Robert Harper.** *Practical Foundations for Programming Languages.* Cambridge, 2012.

30. **Benjamin Pierce.** *Types and Programming Languages.* MIT Press, 2002.

31. **Benjamin Pierce.** *Advanced Topics in Types and Programming Languages.* MIT Press, 2004.

32. **Hendrik Barendregt.** *The Lambda Calculus: its Syntax and Semantics.* Elsevier, 1984.

33. **John Reynolds.** Towards a Theory of Type Structure. *Colloquium sur la programmation.* 1974.

34. **ECMA International.** *ECMAScript Language Specification.* 2011.

35. **David Ungar, Craig Chambers, Bay-Wei Chang, Urs Hölzle.** Organizing Programs Without Classes. *Lisp and Symbolic Computation, Vol. 4*. Kluwer Academic Publishers, 1991.

36. **Glynn Winskel.** *The Formal Semantics of Programming Languages: an Introduction.* MIT Press,

1993.

37. **Guido van Rossum.** *The Python Language Reference, release 3.2.3.* Python Software Foundation, 2012.

38. **Michael Sipser.** *Introduction to the Theory of Computation.* Cengage Learning, 2012.

39. **Ralf Hinze.** Church numerals, twice! *Journal of Functional Programming*, 15(01):1-13, 2005.

40. **Karl Mazurak, Jianzhou Zhao y Stephan Zdancewic.** Lightweight Linear Types in System F. *Types in Language Design and Implementation* págs 77-88. ACM, 2010.

41. **Mozilla.** *The Rust Programming Language.* http://www.rust-lang.org.

42. **Damien Doliguez et al.** Objects in OCaml. *The OCaml system release 4.01: Documentation and user's manual.* http://caml.inria.fr/pub/docs/manual-ocaml-4.01/objectexamples.html.

43. **Bjarne Stroustrup.** *The C++ Programming Language, 4th edition.* Adison Wesley, 2013.

44. **Todd Veldhuizen.** C++ Templates are Turing Complete. 2003.

45. **Gordon Blair, Nelly Bencomo y Robert France.** Models@ run.time. *IEEE Computer 42(10): 22-27.* IEEE, 2009.

46. The OCaml Programming Language. http://ocaml.org/.

47. **Xavier Leroy et al.** Lexer and Parser Generators. *The OCaml System release 4.01, Documentation and User's manual.* http://caml.inria.fr/pub/docs/manual-ocaml-400/manual026.html.

48. **Jon Harrop.** C++ vs OCaml: Ray tracer comparison. http://www.ffconsultancy.com/languages/ray_tracer/comparison.html.

49. **Grigore Rosu.** *K: a Rewrite-based Framework for Modular Language Design, Sematics, Analysis and Implementation.* 2006.

50. **Manuel Clavel, Francisco Durán et al.** All about Maude - A High-Performance Logical Framework, *Lecture Notes in Computer Science 4350*. Springer, 2007.

51. **Grigore Rosu, Wolfram Schultze y Traian-Florin Serbanuta.** Runtime Verification of C Memory Safety. *Runtime Verification 2009*: 132-151.

52. Semantics of C in K. https://github.com/kframework/c-semantics.

53. The semantics of Java in K. https://github.com/kframework/java-semantics.

54. Gérard Berry y Gérard Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1): 217-248. Elsevier, 1992.

55. **Cormac Flanagan y Kenneth Knowles.** Hybrid Type Checking. *TOPLAS*, 32(2). ACM, 2010.

56. **Andrew Sutton y Bjarne Stroustrup.** Design of Concept Libraries for C++. *Software Language*

Engineering 2011, págs 97-118.

57. **Simon Peyton-Jones.** Haskell 98. *Journal of Functional Programming*, 13(1): 0-255. Cambridge University Press, 2003.

58. **Kacper Bak et al.** Example-Driven Modelling: Model = Abstractions + Examples. *Proceedings of the International Conference on Software* Engineering. ACM, 2013.

59. **Jesús Sánchez, Esther Guerra, Juan de Lara.** Bottom-Up Meta-Modelling: An Interactive Approach. *Model Driven Engineering Languages and Systems 2012*, págs 3-19.

60. **Walid Taha.** A Gentle Introduction to Multi-stage Programming. *Domain-Specific Program Generation* 2003, págs 30-50.

Anexo I: Instalación de los compiladores

1. Prerrequisitos

Para obtener y compilar el código fuente de los compiladores desarrollados, son necesarios los siguientes programas:

- Git o Subversion (opcional, se puede emplear la interfaz web de GitHub)
- OCaml (la distribución estándar incluye ocamllex y ocamlyacc)

En Linux: estos programas se pueden encontrar en los repositorios de la mayoría de distribuciones Linux y están disponibles para ser instalados con el gestor de paquetes adecuado (apt-get en Ubuntu). Si se desea obtener la última versión del compilador o instalar OCaml sin disponer de acceso de administrador, se puede emplear ocamlbrew (https://github.com/hcarty/ocamlbrew), que proporciona además un conjunto de utilidades básicas para el desarrollo en OCaml.

En Windows: se puede obtener un instalador de Git para Windows de su página oficial (http://git-scm.com/download/win). La manera recomendada de instalar OCaml es emplear el gestor de paquetes WODI (http://wodi.forge.ocamlcore.org/), que incluye el compilador de OCaml, un gestor de paquetes para descargar extensiones del mismo y un entorno Cygwin básico.

En caso de cualquier problema con la instalación, se puede recurrir a la documentación oficial (http://ocaml.org/docs/install.html) que documenta diversas alternativas de instalación en distintos sistemas.

La versión del compilador de OCaml que se recomienda usar es la 4.01. Sin embargo, no se ha usado ninguna característica avanzada del lenguaje ni ninguna dependencia externa y cabe esperar que funcione con versiones anteriores. Sin embargo, no se recomienda emplear versiones anteriores a la 3.12.

2. Obteniendo el código

Los compiladores de Arche y Archetype se pueden encontrar en los repositorios https://github.com/m-alvarez/Arche y https://github.com/m-alvarez/Archetype respectivamente. Para acceder a los mismos es posible emplear tanto SVN como Git. Por ejemplo, si deseamos obtener el código del compilador de Archetype empleando Git, debemos usar desde la carpeta en la que queramos extraer el

código el comando de consola ‘`git clone https://github.com/m-alvarez/Archetype`’, que creará una carpeta llamada Archetype conteniendo el código del compilador. Para emplear SVN, el comando correcto sería ‘`svn co https://github.com/m-alvarez/Archetype`’, que creará la carpeta Archetype y extraerá el código de la última revisión del compilador en la carpeta Archetype/trunk.

Además de emplear un gestor de versiones, también es posible emplear un navegador web para leer y descargar el código. Para hacer esto, basta dirigirse a la URL del repositorio deseado y emplear la interfaz web para descargar los contenidos del repositorio como un fichero comprimido.

3. Compilando el código

Para compilar cualquiera de los compiladores basta, desde el directorio en el que se ha extraído el código fuente, ejecutar el comando ‘`ocamlbuild arche.byte`’ (para Arche) o ‘`ocamlbuild archetype.byte`’ (para Archetype). Esto generará el fichero ejecutable ‘`arche.byte`’ o ‘`archetype.byte`’ en el mismo directorio. Este ejecutable no es código nativo sino que es bytecode para la máquina virtual empleada opcionalmente por OCaml. Para obtener un ejecutable nativo se pueden emplear los comandos ‘`ocamlbuild arche.native`’ y ‘`ocamlbuild archetype.native`’. El proceso de compilación a código nativo es más lento, pero resulta en un ejecutable más eficiente.

4. Uso de los compiladores

Al ejecutarse sin parámetros, los compiladores proporcionados esperan leer un programa de la entrada estándar y, si el proceso de compilación tiene éxito, imprimen el código JavaScript resultante en la salida estándar. Las opciones de línea de comandos ‘`-in`’ y ‘`-out`’ se pueden emplear para especificar un fichero a ser usado como entrada o salida respectivamente.

Anexo II: Sintaxis de Arche

1. Analizador léxico

A continuación presentamos la estructura léxica del lenguaje Arche, dando el código ocamllex del analizador léxico de su compilador. La sintaxis del mismo es fundamentalmente idéntica a la empleada por el popular analizador léxico flex, y no requiere conocimiento del lenguaje OCaml.

```
{
    open Parser
    exception Unexpected_token of char
}

let ws = [' ' '\t']

rule token = parse
    | ws+ {token lexbuf}
    | '\n'        {Lexing.new_line lexbuf;
                   token lexbuf}
    | "extend"   { EXTEND }
    | "end"      { END }
    | "with"     { WITH }
    | "new"      { NEW }
    | "function" { FUNCTION }
    | ":="        { IS }
    | "{" { LBRACE }
    | "}" { RBRACE }
    | "[" { LBRACKET }
    | "]" { RBRACKET}
    | "(" { LPAR }
    | ")" { RPAR }
    | ";" { SEMICOLON }
    | "," { COMMA }
    | "->" { ARROW }
    | "."  { DOT }
    | "+"  { PLUS }
    | "-"  { MINUS }
    | "*"  { MUL }
    | "/"  { DIV }
    | eof  { EOF }
    | ['a'-'z''A'-'Z''_']+ as lxm { Id(lxm) }
    | ['0'-'9']+ as lxm { Int(int_of_string lxm) }
    | _ as lxm { raise (Unexpected_token lxm) }
```

2. Analizador sintáctico

Presentamos aquí la estructura sintáctica del lenguaje Arche, mostrando el código ocamlyacc empleado para la definición de su gramática. Una vez más, esta gramática resulta accesible sin tener conocimiento de OCaml y es fundamentalmente idéntica a la sintaxis de Bison.

```
%{
    open Ast
%}

%token <string> Id
%token <int> Int
%token EXTEND WITH NEW FUNCTION DELAY IS END
%token LBRACE RBRACE LPAR RPAR LBRACKET RBRACKET MINUS PLUS MUL DIV
%token COMMA COLON SEMICOLON
%left MINUS
%left PLUS
%left MUL
%left DIV
%token EOF
%nonassoc WITH
%nonassoc NEW
%nonassoc DOT
%nonassoc Id
%token DOT ARROW
%nonassoc LPAR RPAR
%start pgm
%type <Ast.expression> pgm

%%

pgm:
    expr EOF { $1 }
;

expr:
    | literal                    { Lit($1) }
    | Id                         { Name($1) }
    | NEW expr                   { New($2) }
    | LPAR expr RPAR             { $2 }
    | EXTEND expr WITH expr      { Extend($2, $4) }
    | expr LPAR expr_list RPAR { Apply($1, $3) }
    | expr DOT Id                { Method($1, $3) }
    | expr op expr               { Bin_op($2, $1, $3) }
;

op:
    | PLUS  { Plus }
    | MINUS { Minus }
    | MUL   { Mul }
    | DIV   { Div }
;

expr_list:
    | expr COMMA expr_list { $1 :: $3 }
    |                      { [] }
;

literal:
    | LBRACE fields RBRACE                         { Obj($2) }
    | FUNCTION LPAR arg_list RPAR ARROW expr END { Fn($3, $6) }
    | Int                                          { Int($1) }
;

arg_list:
    | { [] }
    | Id { [$1] }
    | Id COMMA arg_list { $1 :: $3 }
;
```

```
field:
    | Id pot IS decl SEMICOLON { ($1, { $4 with pot = $2 }) }
;

decl:
    | expr                    { {self=None; pot=0; body=$1} }
    | LPAR Id RPAR expr { {self=Some($2); pot=0; body=$4} }
;

pot:
    |                       { 0 }
    | LBRACKET Int RBRACKET {
        if $2 >= 0
        then $2
        else raise (Failure "Potency must be positive")
    }
;

fields:
    |              { [] }
    | field fields { $1 :: $2 }
;
```

Anexo III: Sintaxis de Archetype

1. Analizador léxico

A continuación presentamos la estructura léxica del lenguaje Archetype, dando el código ocamllex del analizador léxico de su compilador. La sintaxis del mismo es fundamentalmente idéntica a la empleada por el popular analizador léxico flex, y no requiere conocimiento del lenguaje OCaml.

```
{
    open Parser
    exception Unexpected_token of char
}

let ws = [' ' '\t']

rule token = parse
    | ws+ {token lexbuf}
    | '\n'        {Lexing.new_line lexbuf;
                   token lexbuf}
    | "int"      { INT_TYPE }
    | "bool"     { BOOL_TYPE }
    | "begin"    { BEGIN }
    | "end"      { END }
    | "new"      { NEW }
    | "instantiate" { INST }
    | ":="       { IS }
    | "object"   { OBJECT }
    | "inherit" { INHERIT }
    | "{" { LBRACE }
    | "}" { RBRACE }
    | "(" { LPAR }
    | "[" { LBRACKET }
    | "]" { RBRACKET }
    | ")" { RPAR }
    | "+" { ADD }
    | "-" { SUB }
    | "=" { EQ }
    | "if" { IF }
    | "then" { THEN }
    | "else" { ELSE }
    | ";" { SEMICOLON }
    | ":" { COLON }
    | "," { COMMA }
    | "."  { DOT }
    | eof  { EOF }
    | "true" { Bool true }
    | "false" { Bool false }
    | ['0'-'9']+ as lxm { Int(int_of_string lxm) }
    | '-'['0'-'9']+ as lxm { Int(int_of_string lxm) }
    | ['a'-'z''A'-'Z''_']+ as lxm { Id(lxm) }
    | _ as lxm { raise (Unexpected_token lxm) }
```

2. Analizador sintáctico

Presentamos aquí la estructura sintáctica del lenguaje Arche, mostrando el código ocamlyacc empleado para la definición de su gramática. Una vez más, esta gramática resulta accesible sin tener conocimiento de OCaml y es fundamentalmente idéntica a la sintaxis de Bison.

```
%{
    open Ast
%}

%token <string> Id
%token <int> Int
%token <bool> Bool
%token OBJECT
%token INT_TYPE BOOL_TYPE
%token NEW IS INHERIT INST
%token BEGIN END
%token LBRACE RBRACE LPAR RPAR LBRACKET RBRACKET
%token COMMA SEMICOLON COLON
%token ADD SUB EQ
%token IF THEN ELSE
%token EOF
%token DOT
%left ADD SUB
%nonassoc EQ
%nonassoc DOT
%nonassoc ELSE

%start pgm
%type <Ast.program> pgm

%%

pgm:
    def_list BEGIN expr END EOF { ($1, $3) }
;

def:
    Id IS expr SEMICOLON { ($1, $3) }
;

def_list:
    | def def_list { $1 :: $2 }
    | { [] }
;

expr:
    | obj_lit                  { $1 }
    | Id                       { Id($1) }
    | NEW LPAR expr RPAR       { New($3) }
    | LPAR expr RPAR           { $2 }
    | expr DOT Id              { Method($1, $3) }
    | primitive                { Value (Primitive $1) }
    | sugar                    { $1 }
;

num:
    Int { if $1 < 0 then raise (Failure "Number must be positive") else $1 }
;
```

```
sugar:
    | new_sugar { $1 }
;

primitive:
    | int_val   { $1 }
    | int_type  { $1 }
    | bool_val  { $1 }
    | bool_type { $1 }
    | if_expr   { $1 }
    | eq        { $1 }
    | bin_op    { $1 }
;

new_sugar:
    | INST opt_pot_decl LPAR expr COMMA new_sugar_args RPAR {
        let pot    = $2 in
        let parent = $4 in
        let meths  = List.map (function n,m -> n,{m with pot = m.pot + 1}) $6 in
        New(
            InhObj (pot, "self",
                [`Base meths; `Inherit parent]
            )
        )
    }
;

new_sugar_args:
    | Id opt_pot_decl IS expr args_tail {
        ($1, {pot = $2; typ = None; value = Some $4}) :: $5
    }
;

args_tail:
    | { [] }
    | COMMA Id opt_pot_decl IS expr args_tail {
        ($2, {pot = $3; typ = None; value = Some $5}) :: $6
    }
;

if_expr:
    | IF expr THEN expr ELSE expr { Primitives.if_exp $2 $4 $6 }
;

int_val:
    | Int { Primitives.int_lit $1 }
;

int_type:
    | INT_TYPE { Primitives.int_type }
;

bool_val:
    | Bool { Primitives.bool_lit $1 }
;

bool_type:
    | BOOL_TYPE { Primitives.bool_type }
;

eq:
    | expr EQ expr { Primitives.eq $1 $3 }
;
```

```
bin_op:
    | expr ADD expr { Primitives.bin_op `Add $1 $3 }
    | expr SUB expr { Primitives.bin_op `Sub $1 $3 }
;

obj_lit:
    OBJECT level_spec self_spec LBRACE
        fields
    RBRACE {
        let level = $2 in
        let self  = "self" in
        match $5 with
        | [`Base methods] ->
            Value (Obj
                { o_type  = ref None
                ; o_level = level
                ; self    = self
                ; methods })
        | [] ->
            Value (Obj
                { o_type  = ref None
                ; o_level = level
                ; self    = self
                ; methods = [] })
        | _ ->
            InhObj (level, self, $5)
    }
;

self_spec:
    |  { () }
;

level_spec:
    | LBRACKET num RBRACKET { $2 }
    | { 0 }
;

fields:
    | { [] }
    | field SEMICOLON fields {
        match $3 with
        | (`Inherit _) :: _
        | [] ->
            `Base [$1] :: $3
        | (`Base l) :: r ->
            `Base ($1::l) :: r
    }
    | INHERIT expr SEMICOLON fields { (`Inherit $2) :: $4 }
;

field:
    | SUB Id opt_pot_decl opt_type_decl field_body {
        ($2,
        { pot   = $3
        ; typ   = $4
        ; value = $5 })
    }
;

opt_pot_decl:
    | { 0 }
```

```
    | LBRACKET num RBRACKET { $2 }
;

opt_type_decl:
    |             { None }
    | COLON expr { Some $2 }
;

field_body:
    |          { None }
    | IS expr { Some($2) }
;
```