Universidad
Rey Juan Carlos

TESIS DOCTORAL

Diseño y formalización de lenguajes de consultas inspirados en ópticas

Autor:
Jesús López González

Directores:
Juan Manuel Serrano Hidalgo
César Cáceres Taladriz

Programa de Doctorado en Tecnologías de la Información y las Comunicaciones

Escuela Internacional de Doctorado

2020

Diseño y formalización de lenguajes de consultas inspirados en ópticas[1]

Jesús López González

Febrero de 2020

[1]Este trabajo de investigación ha sido parcialmente financiado por una beca del *Programa de Doctorados Industriales* (*Ministerio de Economía y Competitividad del Gobierno de España*) concedida a Habla Computing SL con referencia DI-14-06921.

II

We're more lost than we've ever been
in the wheels of the machine
— The Darkness

Agradecimientos

He de reconocer que siempre me ha llamado especialmente la atención la sección de agradecimientos de toda tesis doctoral, por tratarse de un contenido muy fresco y natural, en la que el pobre doctorando deja entrever su martirio personal. Ahora entiendo que esto es debido a que la sección de agradecimientos es la única que se libra de los cientos de iteraciones que contemplan las correcciones y comentarios de los directores, revisores, colegas, y ya puestos, de uno mismo. Debido a esto, siempre me he imaginado escribiendo esta sección con especial cuidado, pues hay mucha gente a la que agradecer, pero me temo que va a seguir la misma dinámica que el resto de las páginas de este trabajo, con una presión constante por cumplir con los temidos *deadlines* a los que se ve sometido todo doctorando. A pesar de esto, ¡haré de la naturalidad mi estandarte!

En primer lugar y como no podría ser de otra manera, esta tesis le debe mucho, muchísimo, a Juan Manuel Serrano. Juanma ha invertido miles de horas (literalmente) en mi formación, principalmente en forma de conversaciones en las que divagábamos durante horas sobre cuál sería la solución ideal para el problema que se presentaba cada mañana, muchas veces conocedores desde un principio de que su implementación sería inviable. Durante todos estos años he podido apreciar su enorme capacidad técnica, que le permitía enfrentarse a multitud de problemas de forma simultánea o poner en apuros a verdaderas eminencias de un campo que también era nuevo para él, con lo que he podido verificar lo mucho que me queda por aprender. Más allá de estos aspectos, y posiblemente más importante, Juanma se ha convertido en un colega, que ha sabido motivarme y adaptarse en todo momento a mi situación personal. A pesar de esto, me niego en rotundo a aceptar que somos de la misma generación, aunque esto implique que no pueda referirme a este trabajo como *mi primera tesis Chispas*[1]. También me gustaría dar las gracias a César Cáceres por su disponibilidad y por todo el soporte que me ha ofrecido desde la universidad.

Resulta imposible concebir esta tesis sin Isabel de la Morena, que suele referirse a este trabajo de investigación como "un parto muy largo", y no le falta razón. Isa fue la principal responsable de contar conmigo para el proyecto que más tarde sería Habla Computing y de brindarme la oportunidad de realizar la tesis doctoral en este contexto industrial. Aquí, tuvo que sufrir mis inclinaciones académicas, que semana tras semana arruinaban su metódica planificación, por lo que le estoy enormemente agradecido. Y siguiendo por esta línea, me gustaria agradecer a Alberto, Daniel, Javier (Santos), Óscar, Carlos (Fraguas), David, Alberto II de Habla (y V de Alemania), Javier (Fuentes), Ángela, Adrián, Diego, Luis, Giancarlo, Abel, Alfonso, Mikel y Carlos (Vidal), ya que cada uno ha

[1] `https://www.youtube.com/watch?v=Y18jIMMNoHA`

aportado su granito de arena (a veces un castillo entero) en este trabajo.

Y no es que me haya olvidado de Sergio Saugar, sino que se merece su propio párrafo, aunque ya adelanto que tendrá que compartirlo. Sergio fue el tutor de mi Trabajo de Fin de Máster y el que me introdujo en el mundo de Speech con una paciencia y dedicación increíbles, del que también aprendí que esas personas tan lejanas que eran los profesores de universidad podían convertirse en muy buenos colegas. Además es, sin la menor sombra de duda, la persona que mejor se puede poner en mi pellejo, ya que compartimos muchos aspectos de ese martirio personal del que ya voy dando pinceladas. Tanto Sergio como Gema Pérez me dieron consejos muy buenos para abordar esta tesis, que con el paso del tiempo resultan casi obvios, pero que como todo buen doctorando estaba condenado a ignorar.

La calidad de esta tesis se ha visto notablemente mejorada con los comentarios de Jeremy Gibbons, Oleg Kiselyov, Eric Torreborre, Perdita Stevens y los revisores anónimos de los artículos derivados de esta tesis doctoral. Merece una especial atención el agradecimiento a James Cheney, por una revisión verdaderamente detallada sobre el material expuesto en la parte III de este documento. En ella, James nos indicó cómo reorientar el discurso de nuestra investigación (que pasó de "¡muerte a las *for-comprehensions*!" a "¡vivan las *for-comprehension*!" en un par de tardes) y nos mostró la viabilidad de la sección 6.5, que considero fue una pieza esencial para la aceptación del artículo [87]. No puedo olvidarme de Anton Trunov, amo y señor del asistente de pruebas Coq, al que agradezco que me sacara de algún que otro apuro durante la elaboración de las demostraciones formales que se recogen en la parte II de la tesis. Por último, me gustaría dar las gracias a Jesús Medina por informarnos sobre las becas de doctorados industriales, ofrecidas por el Ministerio de Economía y Competitividad, sin la cual este trabajo no hubiera podido realizarse.

Cambiando a un plano más personal, esta tesis y cada éxito académico que he conseguido (ya sea pequeño o casi mediano) no hubiera sido viable sin el apoyo de mis padres y de mi hermana. De vez en cuando oigo hablar sobre *personas hechas a sí mismas* y no puedo evitar poner el concepto en cuestión. Creo firmemente que el entorno es el que te moldea y estoy completamente seguro de que no estaría escribiendo estas líneas si no hubiera sido por el empeño de mi familia en que obtuviera unos estudios. Lamentablemente, es posible que me pasara de frenada, ya que ninguno de mis abuelos va a poder llegar a verme *colocado* en un puesto bueno de una vez por todas (como todos mis primos). Pues bien, ya puedo proclamar con toda certeza que "todavía no, pero ya casi casi", es decir, exactamente la misma respuesta que les daba siempre.

Llegados a este punto, no puedo evitar sentirme como esas personas que llaman a la radio y tienen la necesidad imperiosa de saludar a todo el mundo. En este sentido, me resulta inevitable no mencionar a mis (casi) sobrinos Diego, Martina, Blanca, Gabriel, Abril, Mateo, María y al recién llegado Eren. No obstante, es un saludo con trampa, ya que os obliga a hacer referencia a los artículos que derivan de esta tesis si es que alguna vez llegáis a interesaros por el mundo académico. Antes de tomar esa decisión, permitidme tener con vosotros una conversación larga y tendida en la que trataré de disuadiros de esa idea. No me gustaría terminar este párrafo sin mencionar al resto de mi familia (y familia política), a todos los colegas que a fecha de hoy cuelgan de un tal *burrito sabanero* y al *MPTeam*, por todas las veces que se han interesado por el avance de este trabajo y a los que evidentemente tuve que llamar la atención,

porque *eso nunca se pregunta*.

La realización de una tesis doctoral te cambia la perspectiva sobre el resto del universo, dicho de otra manera, no sales de aquí con todos los jugadores con los que empezaste el partido. Aunque no he llegado al punto de poner a Nela (mi gata) o a Paris y Lambda (mis burras) como coautoras en los trabajos derivados de esta tesis[2], estoy lo suficientemente perjudicado como para incluirlas en mi lista de agradecimientos, por todas aquellas veces que me han permitido desconectar de mis *problemas del primer mundo*, no sin llevarme algún que otro arañazo o coz por el camino. De hecho, como bien sabe Elena, y aludiendo a una referencia cuya fuente es tan patética que no revelaré jamás: "el amor duele".

Y precisamente Elena es la persona a la que va dedicada esta tesis, ya que ha sido quien más ha sufrido con ella, soportando mis cambios de humor que iban a merced del progreso de este trabajo, y a quien ha robado más tiempo. Creo que todas las novedades que nos aguardan a la vuelta de la esquina, entre las que se incluye dar fin a este proyecto (y mucho más complicado, dar fin al alquiler de una carpa), nos llevan directamente a otra nueva etapa de nuestras vidas. Estoy deseando pasarla contigo.

[2] Y créeme cuando te digo que no sería el primero en hacerlo [132].

Índice general

Parte I

Fundamentos básicos

En esta primera parte de la tesis se introducen todos aquellos aspectos que resultan imprescindibles para motivar y poner en contexto el grueso del trabajo. En primer lugar, se describen las alternativas principales que se utilizan en la industria del software para abstraer al programador de la capa de datos: el patrón repositorio *y los* object-relational mappers *(ORMs). Las limitaciones que evidencian estas aproximaciones han propiciado un creciente interés por el* paradigma de programación funcional*, donde existen soluciones más elegantes, como los* repositorios funcionales *—que se nutren del estilo de la* monad transformer library *(MTL)— o las técnicas de* language-integrated query *—que se implementan mediante* lenguajes específicos de dominio *(DSL).*

Estas alternativas también muestran algunos impedimentos y podrían resultar no ser las más adecuadas para lidiar con modelos de datos anidados. De esta manera se llega a la hipótesis de esta tesis: las ópticas*, un conjunto de abstracciones y patrones de composición que se utilizan para manipular estructuras de datos inmutables, podrían inspirar el diseño de lenguajes de consultas que aborden tales problemas. En este sentido, la parte II pretende encontrar analogías entre las ópticas y las abstracciones de MTL, con el objetivo de llevar los beneficios de las primeras a un plano más general; de manera independiente, la parte III identifica el DSL que recoge la esencia de las ópticas y lo utiliza para hacer LINQ. Por lo tanto, el estado del arte recogido en la segunda mitad de esta parte describe en detalle las ópticas, el estilo MTL y la aproximación* tagless-final *para embeber DSLs.*

Capítulo 1

Introducción

Los datos son, sin duda alguna, el elemento más característico que identifica a esta *sociedad de la información* en la que vivimos. Tal es así que se han posicionado como el principal activo de las empresas actuales [74]. Las exigentes demandas de las aplicaciones modernas han derivado en una eclosión de tecnologías muy diversas, centradas en el dato, cuyo fin es el de facilitar el almacenamiento y el procesado de las ingentes cantidades de información que éstas consumen, alcanzando niveles de rendimiento que serían impensables hace no demasiados años atrás. Este tipo de tecnologías suelen incluir lenguajes de consultas para el acceso y la manipulación de la información con la que trabajan. Dada esta situación, resulta muy habitual que la lógica de negocio de la aplicación, implementada en un lenguaje de propósito general (o lenguaje *host*), tenga que interoperar con un lenguaje de consultas (o lenguaje *target*) para poder acceder al estado gestionado por ésta misma. Esta situación puede derivar en ciertas situaciones problemáticas, que se describen en los siguientes párrafos.

En primer lugar, pueden surgir desequilibrios importantes entre el lenguaje host y el lenguaje target llegando a derivar en problemas de mantenibilidad, confiabilidad y seguridad. Uno de los ejemplos más representativos de esta situación es el del *object-relational impedance mismatch* [24, 63], que recoge las dificultades que surgen cuando un lenguaje host basado en el paradigma de orientación a objetos (OO) debe lidiar con un lenguaje target que trabaja sobre una base de datos relacional. Teniendo en cuenta que Java lidera la lista de lenguajes de programación más utilizados en la actualidad[1] y que las tres bases de datos más extendidas a día de hoy son relacionales[2], se puede deducir que es un problema que ha sido objeto de mucha investigación. Prueba de ello es el surgimiento de los *object-relational mappers* (ORMs), que nacían con el objetivo de abstraer al programador OO de los detalles asociados a la capa de persistencia [6]. Desafortunadamente, la abstracción propuesta por un ORM es *leaky* [113], es decir, no es capaz de abstraer los detalles subyacentes en su totalidad. Concretamente, se requiere no sólo que el programador sea consciente de la existencia del ORM —ya que su presencia se hace evidente en el código del lenguaje host— sino también que éste tenga un profundo conocimiento del mismo —a fin de evitar

[1] De acuerdo con el índice TIOBE (`https://www.tiobe.com/tiobe-index/`), donde también se muestra que Java ha ocupado las primeras posiciones desde el año 2000.

[2] De acuerdo con el índice DB-Engines Ranking (`https://db-engines.com/en/ranking`).

acusadas pérdidas de rendimiento.

En segundo lugar, la adaptación a las nuevas tecnologías suele derivar en un indeseado acoplamiento entre la propia tecnología y la lógica de negocio de la aplicación, lo que limita notablemente el mantenimiento y la evolución de la misma. De hecho, el cambio a otra tecnología que gestione el estado de la aplicación o la mera actualización de la ya existente a una versión más moderna son prácticas muy temidas en la industria del software, ya que dicho acoplamiento hace que la complejidad de la tarea sea mucho más elevada que la complejidad asociada al cambio de tecnología en sí mismo. Con el objetivo de evitar este problema, el campo de *Domain-Driven Design* (DDD) [28] propone el concepto de *repositorio*, que se apoya en las interfaces convencionales, y que pretende abstraer al programador de los detalles asociados al acceso a los datos, permitiéndole centrarse de manera exclusiva en la lógica de negocio. Lamentablemente, las interfaces convencionales también son *leaky*, ya que existen numerosas situaciones en las que los detalles particulares de la implementación acaban contaminando la lógica de negocio.

Los problemas descritos anteriormente —que se agravan cuando aparecen de manera conjunta— siguen muy vigentes en la actualidad y se acentúan con la creciente complejidad de los propios sistemas. Esto ha propiciado una incesante búsqueda de nuevas alternativas, lo que ha derivado en que el *paradigma de programación funcional* (PF) haya suscitado un enorme interés por parte de la industria del software durante estos últimos años. Buena cuenta de ello dan la proliferación de lenguajes de programación funcionales (Scala, Haskell, Clojure, etc.), la tendencia a adoptar abstracciones funcionales en lenguajes imperativos [88] y el surgimiento de multitud de conferencias y grupos locales de programación que giran en torno a esta temática. El éxito del paradigma reside en su pureza, carente de *efectos de lado*, que posibilita el *razonamiento ecuacional* del código y que fomenta la aparición de nuevos mecanismos de modularidad [61]. La nueva perspectiva que ofrece este paradigma ha posibilitado la aparición de nuevas aproximaciones para abordar los problemas descritos en los párrafos anteriores, que se introducirán a continuación, que tampoco están exentas de limitaciones.

1.1. Repositorios funcionales

La influencia de la PF también ha llegado al campo del DDD [36]. En particular, la lógica de negocio se nutre de determinadas técnicas funcionales para facilitar la inyección de los repositorios. No obstante, el repositorio propuesto sigue sustentándose en una interfaz convencional, donde únicamente se contempla la posibilidad de que el acceso a la capa de datos pueda producir un error. Esto puede resultar en un inconveniente si se quisieran explotar otros aspectos de la infraestructura subyacente. Por ejemplo, ¿qué pasaría si el estado fuese accesible mediante técnicas de asincronía como *futuros*[3]? Una interfaz limitada a la posibilidad de error no podría beneficiarse de las ventajas derivadas de dicho acceso, ya que la instanciación del repositorio tendría que forzar la sincronía.

En este sentido, los repositorios podrían generalizarse mediante la parametrización del repositorio con un constructor de tipos para abrir la posibilidad a otros efectos, incluyendo, pero no restringiendo el efecto de error. A tal nivel de

[3] `https://docs.scala-lang.org/overviews/core/futures.html`

abstracción resultaría necesario contar con una interfaz que posibilitase la combinación de los diversos programas genéricos, donde encajaría perfectamente la que recoge una *mónada* [92, 126]. Esto nos lleva directamente a un estilo de programación que ha adquirido un enorme interés en la industria del software durante los últimos años.

Las interfaces resultantes, a las que nos referiremos como *repositorios funcionales*, se inspira en el diseño de la librería *monad transformer library* (MTL) [55, 40], que proporciona un catálogo de *teorías algebraicas* [37] para lidiar con efectos computacionales. Estas teorías se implementan como *type classes* [129] que definen una serie de primitivas que suelen tener asociadas un conjunto de leyes. En particular, las interfaces recogidas en MTL (*MonadState*, *MonadError*, etc.) resultan ser de gran utilidad para la compleja tarea de contemplar diversos tipos de computaciones [80] a la vez. En el caso particular de un repositorio parametrizado, suelen desplegarse primitivas ad hoc para acceder a los campos concretos del dominio en cuestión, cuyos resultados pueden ser combinados gracias a la interfaz monádica, aspecto esencial para implementar la lógica de negocio.

No obstante, los repositorios funcionales también tienen asociadas algunas limitaciones que se describen brevemente en los siguientes puntos:

- No existe una manera estándar de definir los repositorios correspondientes a un modelo jerárquico de forma modular y heterogénea. Idealmente, cada entidad del dominio tendría asociado su propio repositorio, donde se recogerían las primitivas de acceso particulares a cada entidad, y la información asociada podría estar desplegada por diversas infraestructuras, por lo que cada repositorio podría estar implementado para lidiar con diferentes tecnologías.
- Los repositorios funcionales tienden a contener primitivas con un patrón que se repite en múltiples ocasiones. Las únicas abstracciones estándar conocidas son las que se recogen en MTL, que pueden no ser las más idóneas para ciertas situaciones. Por ejemplo, no existe ninguna interfaz específica para lidiar con un estado compuesto a partir de una secuencia de valores de un mismo tipo.
- Los repositorios funcionales, y más en general MTL, ofrecen una gran facilidad a la hora de implementar la lógica de negocio de forma general. No obstante, instanciar la pila de efectos que soporte todas las dependencias requeridas puede resultar complejo, ineficiente y en determinadas ocasiones hasta impracticable.

1.2. Language-integrated query

El campo de investigación de *language-integrated query* (LINQ) [117, 90, 19] trata de aliviar los desequilibrios existentes entre lenguajes host y guest mediante una aproximación basada en *lenguajes específicos de dominio* (DSLs) [60, 116]. Desde esta perspectiva, el programador no inyecta consultas codificadas en texto plano sobre el lenguaje de propósito general, lo cual resultaría en una fuente clara de problemas en forma de bugs y vulnerabilidades; por el contrario, el programador utiliza un DSL que asegura que la consulta está bien formada, correctamente tipada y además, reduce el salto existente entre el lenguaje de propósito general y el lenguaje de consultas.

En este sentido, merece la pena destacar que no todo DSL adquiere el sello de aprobación desde una perspectiva de LINQ. Por ejemplo, sería posible embeber el lenguaje SQL en un host como Scala [98] para obtener un tipado seguro y aún así, la disparidad entre los modelos de computación de ambos lenguajes no se habría reducido lo más mínimo. No se niega que este paso sea esencial para construir consultas SQL y buena cuenta dan de ello librerías como [94], que se centran en este aspecto. Sin embargo, para abordar el problema de la impedancia se requieren DSLs a un nivel de abstracción más alto, cercano al lenguaje de propósito general, pero suficientemente específico como para permitir la generación de consultas eficientes [16]. Desde sus inicios más tempranos, el campo de investigación de LINQ ha explotado las *monadic comprehensions* [126, 127] como el DSL por defecto para esta tarea. La primera intuición en esta línea se introdujo en [118], y más tarde fue desarrollada por el *nested relational calculus* (NRC) [10, 11], que establece los fundamentos de los lenguajes de consultas basados en comprehensions. NRC subsume gran parte del trabajo existente sobre teorías y sistemas de LINQ como Kleisli [131], Links [23], Microsoft LINQ [90, 115], Database Supported Haskell (DSH) [122], T-LINQ [19], QUEΛ [114] y SQUR [73].

La idea principal de la investigación en LINQ reside en tomar prestada la sintaxis utilizada por estructuras de datos en memoria tales como listas, conjuntos, bolsas y otras colecciones para componer consultas que trabajen a un nivel genérico. Para esta tarea, estos tipos se *levantan* (*lifting*) en un DSL que abstrae al programador de representaciones específicas pero que mantiene la sintaxis basada en comprehensions. En algunos casos como Kleisli, Links y Microsoft's LINQ, el mecanismo de lifting es una parte primitiva del lenguaje de propósito general en sí mismo. En el caso de lenguajes de programación funcional (FP) más convencionales, como Scala, F#, OCaml, Haskell, etc., el DSL se podría embeber en el lenguaje host utilizando alguna de las técnicas habituales del paradigma funcional para esta misión: typed tagless-final [70], tipos de datos algebraicos generalizados (GADTs) [133, 18] o *quoted* DSLs [93]. Por ejemplo, la técnica de *quotation* se usa para embeber el lenguaje T-LINQ en F# [19], mientras que el estilo tagless-final inspira el diseño de QUEΛ en OCaml. De manera análoga, Quill [9] es un QDSL basado en T-LINQ que se embebe en Scala.

Existen varias ventajas que derivan de la aplicación de estas técnicas para generar consultas específicas en el lenguaje target. En primer lugar, es posible explotar los mecanismos específicos del lenguaje host [16, 73] que no estén disponibles en el lenguaje target. Por ejemplo, a pesar de que SQL no contempla las expresiones lambda como parte de su gramática, es posible explotar los beneficios de esta abstracción en las consultas implementadas desde el lenguaje host, que más tarde serán eliminadas en el proceso de traducción. En segundo lugar, estas traducciones generan consultas específicas muy eficientes, ya que las expresiones originales pueden someterse a optimizaciones muy agresivas mediante técnicas de normalización, que han sido especialmente estudiadas en el caso particular de SQL [23]. Las técnicas de LINQ ponen en cuestión la ley de las *leaky abstractions* [113] —donde se argumenta que toda abstracción introduce *leaks*— o al menos reducen al mínimo su margen de actuación. De hecho, se acuña el eslogan de *abstracción sin culpa* [108, 76, 72], aludiendo a que el enfoque genera el mismo código que produciría manualmente un programador experimentado, pero con todas las ventajas propias de la automatización.

A pesar de todos los beneficios asociados a LINQ, el enfoque habitual basado en comprehensions también presenta algunas limitaciones, descritas a continuación:

- El lenguaje de las comprehensions sólo permite expresar consultas de lectura, pero las actualizaciones son igualmente importantes. Este aspecto se reconoce como un problema abierto en el campo de LINQ [16].
- Las comprehensions están más cerca de la notación *pointwise* propia del cálculo relacional que del algebra relacional pura. Aunque funcionalmente ambos formalismos son igualmente expresivos, los combinadores *point-free* suelen considerarse más flexibles [38]. Este aspecto deriva en consultas más modulares, impactando directamente en aspectos no funcionales como la reutilización y la tolerancia a cambios [104, 61].
- Existen infraestructuras de consultas que son esencialmente jerárquicas en vez de relacionales, como podría ser una base de datos NoSQL orientada a documentos, que se construye sobre fuentes de datos anidadas en JSON, XML o YAML, entre otros. La traducción de las consultas para estas infraestructuras se podría beneficiar de un modelo de consultas más algebraico y jerárquico por naturaleza.

1.3. Ópticas

Las ópticas, también conocidas como *referencias funcionales* [123], son un conjunto de abstracciones que permiten seleccionar *partes* que están contextualizadas en un *todo* para su acceso o manipulación. Por ejemplo, una *lens* [33] es un tipo de óptica que selecciona un valor que está siempre disponible, un *traversal* [95] selecciona una secuencia de partes potencialmente vacía, etc. Cada óptica dispone de una interfaz específica para la evolución del estado mediante la evolución de las partes que selecciona. Es habitual que estas interfaces tengan asociadas unas leyes que las instancias deben cumplir. Por ejemplo, si se observa la parte y justo después se actualiza dicha parte con el resultado obtenido, no se debería de apreciar ningún cambio en el todo.

Desde la primera aparición del concepto de *lens*, sin duda alguna la óptica más popular, ha surgido un rico catálogo de ópticas que se muestra en la figura 1.1[4], donde las diversas abstracciones forman una jerarquía en la que las flechas determinan las posibles traducciones entre ellas. Si se tiene en cuenta que toda óptica tiene una composición cerrada[5], esta jerarquía esta realmente determinando las posibilidades de composición heterogénea de las ópticas. Por ejemplo, la combinación de una *Lens* con un *Prism* dará como resultado un *AffineTraversal*, por tratarse del ancestro común. Siguiendo esta línea de pensamiento, no sería posible combinar un *Fold* con un *Setter*.

La diversidad de abstracciones y la composicionalidad de éstas posibilita la composición de consultas que pueden alcanzar un alto grado de complejidad de manera concisa y elegante. Consecuentemente, estas técnicas han gozado de una

[4]Tomada prestada de un post de Oleg Grenrus donde que describe Glassery [43] que es, según nuestro conocimiento, la librería de ópticas más completa hasta la fecha.

[5]Es decir, se pueden componer dos instancias de la misma óptica, siempre que los tipos lo permitan.

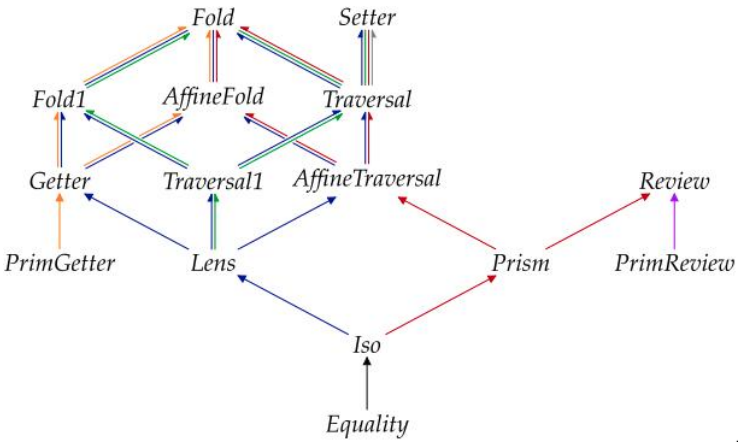

Figura 1.1: Jerarquía de ópticas

gran popularidad entre la comunidad de programadores funcionales, tal y como se refleja en la eclosión de librerías industriales basadas en ópticas durante la última década [75, 119, 96, 43, 107]. De hecho, estas abstracciones han adquirido tal notoriedad que pueden encontrarse en el código de productos tan extendidos como el videojuego de construcción Minecraft de Microsoft[6].

Desafortunadamente, las ópticas están restringidas a la manipulación de estructuras de datos en memoria, mientras que el estado con el que trabajan los sistemas actuales se dispone en bases de datos, servicios web, cachés, etc. tal y como se apuntaba al principio de esta introducción. En este contexto, la aproximación basada en ópticas no resulta inservible, pero está severamente limitada. En términos generales, se requeriría la previa carga de la información a estructuras de datos en memoria, lo que suele resultar en una práctica poco eficiente o muy posiblemente inviable.

1.4. Hipótesis, objetivos y estructura de este documento

Una vez introducidos los principales objetos de interés (repositorios, LINQ y ópticas) y algunas de las limitaciones asociadas a cada uno, se propone la siguiente hipótesis para iniciar la investigación de esta tesis doctoral:

Hipótesis *Es posible diseñar lenguajes de consultas que se inspiren en las abstracciones y patrones de composición propios de las ópticas, permitiendo ampliar el rango de actuación de éstas más allá de las estructuras de datos inmutables en memoria. Los lenguajes resultantes deberían de mitigar los problemas asociados a los repositorios funcionales y a las soluciones LINQ basadas en comprehensions descritos previamente.*

[6] `https://www.reddit.com/r/haskell/comments/9m2o5r/digging_reveals_profunctor_optics_in_mineacraft/`

Objetivos

La hipótesis establecida permite orientar la investigación en dos líneas de trabajo independientes, en torno a las cuales se definen los objetivos. Antes de llegar a ellos, se motivarán muy brevemente los pilares sobre los que se fundamentan cada una de estas líneas.

Por un lado, existe un parecido considerable entre la óptica *Lens* y la teoría algebraica recogida en MTL que se conoce como *MonadState*, tanto por las primitivas que componen sus interfaces como por las leyes asociadas. Sin embargo, es evidente que *MonadState*, por tratarse de una teoría algebraica de MTL, trabaja a un nivel de abstracción más general. Esta relación plantea diversas preguntas de investigación: ¿cuál es la conexión precisa que existe entre ambas abstracciones?, ¿existen otras teorías algebraicas que se correspondan con otras ópticas?, ¿exhibe *MonadState* un mecanismo de composición análogo al que muestra una *Lens*?, etc. En definitiva, la relación entre MTL y las ópticas parece ser un marco interesante que establece la primera línea de investigación.

Por otro lado, en vez de buscar analogías entre ópticas y abstracciones genéricas ya existentes, se puede optar por una vía más directa: hacer un *lifting* de las primitivas existentes en las librerías de ópticas actuales y empaquetarlas en un DSL. Esto tendría una fuerte correspondencia con las técnicas existentes de LINQ. De hecho, también se estaría utilizando una sintaxis muy extendida como es la de las ópticas para lidiar con estructuras de datos inmutables, pero en un plano más general. De nuevo, se plantean nuevas preguntas de interés: ¿qué dominios semánticos soportaría este lenguaje?, ¿se podrían llevar a cabo optimizaciones durante la traducción?, ¿cómo se relacionaría esta aproximación con las técnicas de LINQ basadas en comprehensions?, etc. Diseñar un nuevo lenguaje basado en las ópticas para hacer LINQ se postula como segunda línea de investigación.

El enfoque industrial de esta tesis requiere que los resultados obtenidos se lleven a un plano industrial, o al menos que se presenten como una prueba de concepto que apunte hacia esta dirección. Por ello, la implementación de las librerías que pongan en práctica las ideas que derivan de esta investigación en un lenguaje de ámbito industrial es un requisito fundamental que debe ser compartido por ambas líneas de trabajo. Con todo esto, se establecen los siguientes objetivos para esta tesis doctoral:

Obj. 1. Formalizar la conexión existente entre *Lens* y *MonadState*; determinar una noción de composición para *MonadState* en el plano genérico; determinar el proceso de diseño que permite identificar teorías algebraicas análogas a otras ópticas y aplicar estos mismos pasos sobre las nuevas conexiones (capítulo 3).

Obj. 2. Implementar las ideas del punto anterior en un lenguaje de aplicación industrial como Scala y experimentar con el proceso de diseño resultante del objetivo anterior para ampliar el catálogo de abstracciones; identificar patrones de diseño y estructuras recurrentes que surgen durante el desarrollo de aplicaciones; mostrar la generalidad de la aproximación ofreciendo implementaciones experimentales para diversas infraestructuras (capítulo 4).

Obj. 3. Diseñar un lenguaje de consultas que se inspire en las abstracciones y combinadores habituales en las librerías de ópticas existentes, destinado a

hacer LINQ; formalizar la semántica estándar del lenguaje y semánticas no estándar para acceder a otras fuentes de datos con unas garantías mínimas de eficiencia; establecer la relación existente con las aproximaciones de LINQ basadas en comprehensions (capítulo 6).

Obj. 4. Embeber el lenguaje diseñado en el punto anterior en un lenguaje de ámbito industrial; validar la aproximación mediante la traducción de consultas genéricas a consultas específicas para diversas infraestructuras (capítulo 7).

Esencialmente, la parte II de esta tesis recoge los objetivos 1 y 2, mientras que la parte III desarrolla los objetivos 3 y 4. Por su parte, los capítulos 5 y 8 discuten de manera independiente los resultados obtenidos en las partes II y III, respectivamente. La parte IV concluye este trabajo (capítulo 9) apuntando hacia trabajo futuro. Por último, la parte I, adicionalmente a este capítulo introductorio, también ofrece un estudio del estado del arte que describe en mayor grado de detalle las ópticas, el estilo MTL, los repositorios funcionales, la aproximación tagless-final y LINQ mencionados anteriormente (capítulo 2). Merece la pena destacar los apéndices, donde se ofrece una breve introducción al lenguaje de programación Scala (apéndice A), que será utilizado para guiar las explicaciones, y donde también se muestra una sección que describe el contexto de descubrimiento (apéndice B), donde se describen los principales hitos e impedimentos encontrados durante el desarrollo de esta tesis doctoral, siguiendo un orden temporal y adoptando un tono más informal.

Capítulo 2

Estado del arte

Este capítulo allana el terreno hacia los contenidos principales de esta tesis, describiendo las técnicas en las que se apoya en mayor grado de detalle. En primer lugar se introducen las diferentes abstracciones que caen bajo el paraguas de las ópticas (sección 2.1), los patrones de diseño que surgen en torno a ellas y diferentes representaciones o codificaciones del concepto. Se describen también MTL (sección 2.3) y tagless-final (sección 2.4), las técnicas en las que se sustentan la parte II y la parte III de este trabajo, respectivamente, para el diseño de lenguajes inspirados en ópticas. Para complementar esta sección, se detallan los problemas que surgen con los repositorios (sección 2.2) y con las técnicas de LINQ basadas en comprehensions (sección 2.5).

A lo largo del documento se utilizará Scala como lenguaje de programación principal para guiar las explicaciones[1]. El apéndice A ofrece un breve background para ilustrar los elementos del lenguaje que se consideran más relevantes en este contexto. Más allá de la adopción del plugin *kind-projector*, cuyo uso preciso queda plasmado en dicho apéndice, no se supondrá ninguna extensión adicional al lenguaje. No obstante, sí que se asumirá la existencia de una inferencia de tipos más potente[2], con el fin de simplificar las definiciones y hacerlas más legibles al lector.

Observación 1. El hecho de que MTL y tagless-final se introduzcan de manera independiente puede confundir al lector, ya que MTL suele entenderse como una instancia particular de tagless-final. No obstante, cada parte de este trabajo explota aspectos muy diferentes de cada aproximación, que se enfatizarán en estas secciones. Se arrojará más luz sobre esta discusión más adelante, según se vaya considerando necesario.

2.1. Ópticas

Las ópticas, también conocidas como *referencias funcionales* [123], son un conjunto de abstracciones que permiten la selección de *partes* contextualizadas

[1] Excepto en el capítulo 3, donde Coq se utilizará para introducir las definiciones que se explotan en las pruebas formales.

[2] Existen determinadas situaciones en las que el compilador de Scala no es capaz de inferir los tipos de una expresión correctamente, especialmente cuando existen constructores de tipos de por medio.

en un *todo*, proporcionando métodos para acceder *y/o* actualizar los valores que forman parte de la selección. Las ópticas suelen venir acompañadas por un conjunto de leyes que definen el correcto comportamiento de dichos métodos. No obstante, este trabajo únicamente presta atención a las leyes asociadas a *lens*, la primera óptica conocida [33] y también la que goza de mayor popularidad. Desde la primera aparición del concepto de lens ha surgido un amplio catálogo de ópticas que se ilustra en la figura 1.1. Esta sección pretende introducir las ópticas más relevantes en el contexto de esta tesis y describir las representaciones más habituales para éstas. También se mostrará trabajo relacionado donde las lenses se combinan con efectos. Finalmente, la sección 2.1.4 muestra las ópticas como un lenguaje de consultas, donde se ilustrará el potencial de estas abstracciones por medio de varios ejemplos.

2.1.1. Catálogo de ópticas

Se introducen las ópticas que se consideran más relevantes en el contexto de este trabajo: *lens*, *affine traversal*, *traversal*, *getter*, *affine fold* y *fold*. Para cada una de ellas se mostrará una breve definición y los combinadores que se utilizarán a lo largo de esta tesis. Esta sección utiliza la representación concreta en las definiciones, por fines didácticos, a excepción de traversal donde se utiliza una representación diferente por motivos que se detallarán más adelante. La sección 2.1.2 muestra representaciones alternativas, aunque se centra en el caso particular de lens.

Definición 1 (Lens)**.** Una *lens* es una óptica que selecciona una única parte contextualizada en un todo y permite su lectura y actualización.

```
case class Lens[S, A](get: S => A, set: A => S => S)
```

Los parámetros tipo S y A se usarán de forma consistente para referirnos al todo y a la parte, respectivamente, en esta y en definiciones sucesivas. Las leyes asociadas a esta definición se recogen en la sección 2.1.2.

La figura 2.1 muestra algunos combinadores básicos para lenses. En particular, el combinador `andThen` permite la selección de partes a un nivel de anidamiento más profundo mediante la composición de selecciones más sencillas. El constructor `id` actúa como elemento neutro en la composición introducida por `andThen`. Se incluye un bloque de sintaxis que introduce $\ggg$ como versión infija para el combinador `andThen`, es decir, `ln1` $\ggg$ `ln2` es equivalente a `andThen(ln1, ln2)`. El método `andThen` es recurrente en todas las ópticas (ya que todas las ópticas son cerradas bajo composición) por lo que la sintaxis $\ggg$ se asumirá para todas las ópticas, aunque no se indique por brevedad.

Observación 2. La implementación propuesta para métodos como `andThen` no resulta idiomática en Scala. En su lugar, debería haberse implementado como un método de la propia clase, donde la noción de *this* (propia del paradigma OO) estaría disponible:

```
case class Lens[S, A](get: S => A) {
  def andThen[B](other: Getter[A, B]): Getter[S, B] = ...
}
```

Sin embargo, tal y como se verá más adelante, existen algunos combinadores donde `A` debe ser un tipo muy específico. Implementar estos combinadores re-

queriría de evidencias *Leibniz*[3] para su implementación. A fin de evitar esta complejidad y poniendo el foco en la homogeneidad entre los combinadores se decide mantener la versión actual, añadiendo sintaxis adicional (como ⋙) que permita que al menos las invocaciones sí que resulten idiomáticas. Como ganancia adicional, esta versión guarda más parecido con lo que se mostrará en el capítulo 6.

```
object Lens {

  def id[A]: Lens[A, A] = Lens(identity, const)

  def andThen[S, A, B](ln1: Lens[S, A], ln2: Lens[A, B]): Lens[S, B] =
    Lens(
      s => ln2.get(ln1.get(s)),
      b => s => ln1.set(ln2.set(b)(ln1.get(s)))(s))

  trait Syntax {
    implicit class LensInfix[S, A](ln: Lens[S, A]) {
      def ⋙[B](other: Lens[A, B]): Lens[S, B] = andThen(ln, other)
    }
  }

  object syntax extends Syntax
}
```

Figura 2.1: Algunos combinadores para lenses.

Definición 2 (Affine Traversal)**.** Un *affine traversal* es una óptica que selecciona como máximo una parte contextualizada en un todo. Permite tanto la lectura como la actualización de ésta, en caso de que exista.

```
case class AffineTraversal[S, A](preview: S => Option[A], set: A => S => S)
```

En la figura 2.2 se recogen los combinadores básicos de un affine traversal, donde de nuevo aparecen `id` y `andThen`. Su implementación es muy similar a la de los combinadores análogos para lens, sólo que en esta ocasión es necesario lidiar con la posibilidad de que la parte no aparezca seleccionada. Este aspecto se manifiesta en ambos combinadores: `id` envuelve la parte un valor opcional definido (`Some`); `andThen` utiliza una *for-comprehension* sobre `Option` para efectuar la lectura y requiere una introspección de la parte seleccionada para poder definir la operación de escritura.

Observación 3. Uno de los principales atractivos de las ópticas es que se pueden componer de forma heterogénea; en otras palabras, es posible combinar ópticas de distinto tipo como lenses, getters, etc. Por ponerlo bajo otra perspectiva, es posible traducir unas ópticas en otras, siguiendo las flechas introducidas en la figura 1.1. Un ejemplo de este tipo de casting se muestra en la figura 2.2 ($\texttt{to}_{ln}$), donde aparece implementado como un conversor implícito. Por tanto, el compilador de Scala aplicará esta función en aquellas situaciones en las que se detecte una óptica de tipo lens pero esperaba encontrarse con un affine traversal. Como puede resultar obvio, esta implementación se corresponde con la flecha que une lens con affine traversal.

```
object AffineTraversal {

  def id[A]: AffineTraversal[A, A] = AffineTraversal(a => Some(a), const)

  def andThen[S, A, B](
      af1: AffineTraversal[S, A],
      af2: AffineTraversal[A, B]): AffineTraversal[S, B] =
    AffineTraversal(
      s => for { a ← af1.preview(s); b ← af2.preview(a) } yield b,
      b => s => af1.preview(s).fold(s)(a => af1.set(af2.set(b)(a))(s)))

  implicit def to_ln[S, A](ln: Lens[S, A]): AffineTraversal[S, A] =
    AffineTraversal(s => Some(ln.get(s)), ln.set)
}
```

Figura 2.2: Algunos combinadores para affine traversals.

Definición 3 (Traversal)**.** Un *traversal* es una óptica que selecciona una secuencia de partes (posiblemente vacía) contextualizada en un todo. Permite tanto la lectura como la actualización de las mismas.

```
trait Traversal[S, A] {
  def apply[F[_]: Applicative](f: A => F[A]): S => F[S]
}
```

La definición anterior requiere una explicación más detallada, ya que su codificación resulta bastante peculiar. De hecho, una representación sencilla basada en un método de lectura y otro de escritura no resulta viable para esta óptica. Como aproximación inicial, uno podría identificar la siguiente abstracción para seleccionar una secuencia de partes que contemple su lectura y actualización:

```
case class Traversal[S, A](getAll: S => List[A], setAll: List[A] => S => S)
```

El problema reside en que esta abstracción no establece ninguna consistencia entre `getAll` y `setAll`. Por ejemplo, nada previene al programador de proporcionar a `setAll` una secuencia de partes con una cardinalidad diferente a la que produce `getAll` cuando ambos trabajan con el mismo todo. Este aspecto podría arreglarse con una representación que hiciera uso de *tipos existenciales* [105], pero hay alternativas más sencillas para un lenguaje como Scala. En particular, la representación que se muestra en la definición 3 se conoce como *van Laarhoven* [95] y se caracteriza porque preserva estas condiciones por defecto[4]. La figura 2.3 muestra los combinadores habituales de `Traversal`. Merece la pena destacar la implementación tan elegante de `andThen`, que permite vislumbrar las ventajas de composición ofrecidas por esta representación. Por último, se recoge $\texttt{to}_{at}$, que ofrece una implementación para la flecha que une el affine traversal con el traversal en la figura 1.1.

Definición 4 (Getter)**.** Un *getter* consiste en una función que selecciona una única parte contextualizada en un todo, de la que sólo se permite su lectura.

```
case class Getter[S, A](get: S => A)
```

[3] https://github.com/scalaz/scalaz/blob/e16bbb2d2a2acd6a5cebd1d0dca7fcf67fae7a78/core/src/main/scala/scalaz/Leibniz.scala

[4] La sección 2.1.2 mostrará la representación van Laarhoven de una lens en detalle, lo que ayudará a entender la codificación tan particular propuesta aquí.

```
object Traversal {
  import Applicative.syntax._

  def id[A]: Traversal[A, A] = new Traversal[A, A] {
    def apply[F[_]: Applicative](f: A => F[A]): A => F[A] = f
  }

  def andThen[S, A, B](
      tr1: Traversal[S, A],
      tr2: Traversal[A, B]): Traversal[S, B] =
    new Traversal[S, B] {
      def apply[F[_]: Applicative](f: B => F[B]): S => F[S] = tr1(tr2(f))
    }

  implicit def to_at[S, A](at: AffineTraversal[S, A]): Traversal[S, A] =
    new Traversal[S, A] {
      def apply[F[_]: Applicative](f: A => F[A]): S => F[S] =
        s => at.preview(s).fold(pure(s))(a => pure(at.set(a)(s)))
    }
}
```

Figura 2.3: Algunos combinadores para traversals.

Por tanto, un getter puede entenderse como una lens sin capacidades de escritura.

Existe un gran catalogo de combinadores estándar asociados a los getters. En la figura 2.4 se han seleccionado aquellos que se usarán de forma frecuente a lo largo de este trabajo, donde aparecen recogidos en el *companion object* que acompaña a la clase `Getter`. El método `andThen` permite combinar getters que seleccionan valores anidados con el objetivo de construir nuevos getters que seleccionan partes anidadas a varios niveles de profundidad. El getter `id` es el elemento neutro en términos de la composición `andThen`, donde el todo y la parte coinciden por lo que podríamos entenderlo como la selección de sí mismo. El combinador `fork` es necesario para juntar diferentes selecciones en una sola. El combinador `like` selecciona una constante que se recibe como parámetro, donde el todo es completamente ignorado. El resto de combinadores levantan operaciones aritméticas y las convierten en getters. Para ello, suelen tomar como parámetros uno o varios getters que seleccionan los operandos para producir un getter que selecciona el resultado de la operación. Por último, $\texttt{to}_{ln}$ evidencia la posible traducción entre una lens y un getter.

Observación 4. Se asumirá $\star\star\star$ como la versión infija para el combinador `fork`, símbolo que tiene precedencia sobre $\ggg$. Además, se sobrecargan los operadores `+++`, `>`, y `-` como versiones infijas para `equal`, `greaterThan` y `subtract`, respectivamente. Por último, se usará la expresión postfija `p.not` como un alias para `not(p)`. No se muestra la extensión para la nueva sintaxis, aunque la versión completa puede encontrarse en el soporte digital que acompaña a esta tesis.

Observación 5. La composición introducida por `fork`, a la que también nos referiremos como *composición horizontal*[5] no es tan extendida en el folclore. De hecho, no es posible implementarla de forma segura para muchas ópticas. Por ejemplo, la implementación análoga de un combinador que realice combinación

[5]Y consecuentemente, nos referiremos al combinador `andThen` como *composición vertical*.

```
object Getter {

  def id[A]: Getter[A, A] = Getter(a => a)

  def andThen[S, A, B](u: Getter[S, A], d: Getter[A, B]): Getter[S, B] =
    Getter(s => d.get(u.get(s)))

  def fork[S, A, B](l: Getter[S, A], r: Getter[S, B]): Getter[S, (A, B)] =
    Getter(s => (l.get(s), r.get(s)))

  def like[S, A](a: A): Getter[S, A] = Getter(const(a))

  def not[S](b: Getter[S, Boolean]): Getter[S, Boolean] = b ⋙
Getter(!_)

  def equal[S, A](x: Getter[S, A], y: Getter[S, A]): Getter[S, Boolean] =
    Getter(s => x.get(s) == y.get(s))

  def greaterThan[S](x: Getter[S, Int], y: Getter[S, Int]): Getter[S, Boolean] =
    Getter(s => x.get(s) > y.get(s))

  def subtract[S](x: Getter[S, Int], y: Getter[S, Int]): Getter[S, Int] =
    Getter(s => x.get(s) - y.get(s))

  implicit def to_ln[S, A](ln: Lens[S, A]): Getter[S, A] =
    Getter(ln.get)
}
```

Figura 2.4: Algunos combinadores para getters.

horizontal sobre lenses violaría sus leyes [31] cuando ambas lenses seleccionen la misma parte del todo.

Definición 5 (AffineFold)**.** Un *affine fold* consiste en una función que selecciona como máximo una parte contextualizada en un todo, de la que sólo se permite su lectura.

```
case class AffineFold[S, A](preview: S => Option[A])
```

Por tanto, un affine fold puede entenderse como un affine traversal sin capacidades de escritura.

De nuevo, se han recopilado varios combinadores que producen affine folds en la figura 2.5. El combinador `id` simplemente selecciona el todo y lo envuelve en un `Some`, uno de los posibles casos del tipo de datos algebraico `Option`. El combinador `andThen` selecciona un valor a varios niveles de anidamiento siempre y cuando las selecciones de `u` y `d` estén definidas; en caso contrario no seleccionará ninguna parte, es decir, seleccionará `None`. Esta funcionalidad se implementa utilizando sintaxis de *for-comprehension* sobre la mónada `Option`. El combinador `filtered` es un caso especialmente interesante: se encarga de envolver al todo en un `Some` siempre y cuando se cumpla el predicado sobre él; en caso contrario, si el predicado no se cumple, se filtrará la selección y se devolverá `None`. Por último, y de acuerdo con la jerarquía de ópticas, $\texttt{to}_{gt}$ y $\texttt{to}_{at}$ implementan la traducción de getters y affine traversals en affine folds.

```
object AffineFold {

  def id[A]: AffineFold[A, A] = AffineFold(a => Some(a))

  def andThen[A, B, C](
      u: AffineFold[A, B],
      d: AffineFold[B, C]): AffineFold[A, C] =
    AffineFold(s =>
      for {
        b ← u.preview(s)
        c ← d.preview(b)
      } yield c)

  def filtered[S](p: S => Boolean): AffineFold[S, S] =
    AffineFold(s => if (p(s)) Some(s) else None)

  implicit def to_gt[S, A](g: Getter[S, A]): AffineFold[S, A] =
    AffineFold(s => Some(g.get(s)))

  implicit def to_at[S, A](at: AffineTraversal[S, A]): AffineFold[S, A] =
    AffineFold(at.preview)
}
```

Figura 2.5: Algunos combinadores para affine folds.

Definición 6 (Fold)**.** Un *fold* consiste en una función que selecciona una secuencia de partes, potencialmente vacía, contextualizadas en un todo, de las que sólo se permite su lectura.

```
case class Fold[S, A](getAll: S => List[A])
```

Por tanto, un fold puede entenderse como un traversal sin capacidades de escritura.

Una vez más, se recopilan algunos combinadores relacionados con folds en la figura 2.6. La implementación de `id` y `andThen` es esencialmente la misma que se ha propuesto para los combinadores análogos de los affine folds, pero la diferencia radica en que en esta ocasión se trabaja con la mónada `List` en lugar de la mónada `Option`[6]. El método `nonEmpty` toma un fold como parámetro y devuelve un getter que selecciona **`true`** si el argumento selecciona al menos una parte o **`false`** en caso contrario. El resto de combinadores (`empty`, `all`, `any` y `elem`) son definiciones derivadas, es decir, se implementan a partir de otros combinadores, donde se asume que la sintaxis de orientación a objetos está disponible. Por ejemplo, `nonEmpty(fl)` se convierte en `fl.nonEmpty` y `all(fl)(p)` se convierte en `fl.all(p)`. Finalmente, $\texttt{to}_{tr}$ y $\texttt{to}_{af}$ implementan las flechas de la jerarquía que unen traversal y affine fold con la óptica fold.

Observación 6. Getter, affine fold y fold son ejemplos de *read-only optics*. Este subconjunto de ópticas ofrecen mecanismos para observar las partes seleccionadas pero no contemplan su actualización y por tanto no están sujetas a leyes que deban preservar la consistencia entre lecturas y escrituras [43, 31]. Las ópticas de sólo lectura no están tan extendidas como sus compañeras con capacidades de actualización (lens, traversal, etc.) dado que la lectura de partes anidadas en

[6]De hecho, también se podría haber utilizado la misma implementación en los getters, utilizando la mónada `Id`, pero esto se ha evitado por simplicidad.

```
object Fold {

  def id[A]: Fold[A, A] = Fold(a => List(a))

  def andThen[A, B, C](u: Fold[A, B], d: Fold[B, C]): Fold[A, C] =
    Fold(s =>
      for {
        b ← u.getAll(s)
        c ← d.getAll(b)
      } yield c)

  def nonEmpty[S, A](fl: Fold[S, A]): Getter[S, Boolean] =
    Getter(fl.getAll(_).nonEmpty) /* List.nonEmpty */

  def empty[S, A](fl: Fold[S, A]): Getter[S, Boolean] =
    fl.nonEmpty.not

  def all[S, A](fl: Fold[S, A])(p: A => Boolean): Getter[S, Boolean] =
    (fl ⋙ filtered[A](a => !p(a))).empty

  def any[S, A](fl: Fold[S, A])(p: A => Boolean): Getter[S, Boolean] =
    fl.all(a => !p(a)).not

  def elem[S, A](fl: Fold[S, A])(a: A): Getter[S, Boolean] =
    fl.any(_ == a)

  implicit def to_tr[S, A](tr: Traversal[S, A]): Fold[S, A] =
    Fold(s => tr[λ[x => List[A]]](a => List(a))(constApplicative[A])(s))

  implicit def to_af[S, A](a: AffineFold[S, A]): Fold[S, A] =
    Fold(s => a.preview(s).toList)
}
```

Figura 2.6: Algunos combinadores para folds.

estructuras de datos inmutables resulta ser una tarea trivial, aunque serán determinantes en este trabajo. En cualquier caso, estas ópticas exhiben las mismas capacidades de composición y patrones de diseño propios del resto de ópticas, tal y como se mostrará en la sección 2.1.4.

Observación 7. Las ópticas de sólo lectura consisten en meras funciones que seleccionan partes contextualizadas en un todo. A pesar de esto, se ha hecho mucho énfasis en introducir estas ópticas como definiciones independientes. La distinción entre funciones y ópticas resulta una pieza clave en este trabajo, ya que las expresiones del lenguaje Optica denotando ópticas o funciones podrían ser evaluadas de forma muy diferente, tal y como veremos en el capítulo 6.

2.1.2. Representaciones alternativas de lens

Existen múltiples formas de codificar una lens [125, 111, 105]. Cada una de ellas pone énfasis en un aspecto determinado: composicionalidad, simplicidad, etc. En esta sección introduciremos la versión basada en un morfismo de mónada estado, la versión basada en profunctores, la versión comúnmente conocida como *van Laarhoven* [95][7] y la versión basada en la comonad coalgebra de coestado, por ser las más relevantes en relación con el trabajo que aquí se presenta. Antes de esto, se revisitará la versión concreta presentada en la definición 1, que será extendida con material nuevo.

Observación 8. La definición 1 se corresponde con la de una lens monomórfica, donde la abstracción no soporta cambios en los tipos asociados al todo y a la parte, es decir, `S` y `A` se preservan durante la evolución de la estructura de datos inmutable. Sin embargo, algunas ópticas soportan la noción de actualización polimórfica [97], donde se puede configurar un nuevo tipo `B` para la parte, lo que también deriva en la configuración de un nuevo tipo `T` para el todo. Así es como surgen los característicos parámetros *stab* (`LensP[S, T, A, B]`) que identifican a una lens en muchas librerías. Esta sección pretende mostrar el potencial de cada representación, por lo que el soporte de la representación para la variante polimórfica es un aspecto a tener en cuenta. En este sentido, es importante destacar que una lens polimórfica subsume a una lens monomórfica, es decir, `type Lens[S, A] = LensP[S, S, A, A]`.

Revisitando la representación concreta

La representación concreta polimórfica de una lens puede codificarse de la siguiente manera:

```
case class Lens[S, T, A, B](get: S => A, set: B => S => T)
```

La adaptación debería resultar sencilla, si se toma la definición 1 como referencia. Aunque este aspecto fue descuidado durante la sección 2.1.1, las lenses tienen asociadas una serie de leyes [31]. En particular, las siguiente propiedades definen una *very well-behaved lens*[8]:

[7] Esta representación se denomina *transformer* en [125], pero evitamos este nombre, ya que podría crear confusión con la representación basada en morfismos de mónadas, muy cercanos a los transformadores de mónadas.

[8] Las leyes asociadas a la versión polimórfica no parecen estar muy claras en la actualidad, tal y como sugiere este thread `https://github.com/julien-truffaut/Monocle/issues/430`. Por tanto, se definen en torno a la versión monomórfica (`Lens[S, S, A, A]`).

```
def getSet[S, A](ln: Lens[S, S, A, A], s: S) =
  ln.set(ln.get(s))(s) == s

def setGet[S, A](ln: Lens[S, S, A, A], s: S, a: A) =
  ln.get(ln.set(a)(s)) == a

def setSet[S, A](ln: Lens[S, S, A, A], s: S, a1: A, a2: A) =
  ln.set(a2)(ln.set(a1)(s)) == ln.set(a2)(s)
```

La propiedad `getSet` indica que la actualización de la parte con el valor actual no debería producir ningún cambio en el todo; la propiedad `setGet` postula que la lectura de la parte debería ser consistente con una actualización previa; la propiedad `setSet` indica que el resultado de encadenar varias actualizaciones debería ser igual que el de llevar a cabo únicamente la última actualización. Finalmente, merece la pena destacar que descartando `setSet` del conjunto de propiedades, se llega a la noción más relajada de *well-behaved lens*.

Observación 9. Aunque este trabajo de investigación se centra únicamente en las leyes que acompañan a una lens, se recuerda que muchas otras ópticas también tienen un conjunto de leyes asociado a su definición. El lector interesado puede encontrar las leyes asociadas a otras ópticas en la primera parte de [84], una serie de posts que derivan directamente del trabajo realizado en esta tesis. Las ópticas de sólo lectura no tiene leyes asociadas, como ya se apuntó en la observación 6.

La clase de lenses (very) well-behaved tiene una composición cerrada, que se implementa con la siguiente definición:

```
def andThen[A, B, C, D, E, F](
    ln1: Lens[A, B, C, D],
    ln2: Lens[C, D, E, F]): Lens[A, B, E, F] =
  Lens(
    a => ln2.get(ln1.get(a)),
    f => a => ln1.set(ln2.set(f)(ln1.get(a)))(a))
```

Aquí se puede apreciar que es posible componer dos lenses para formar otra nueva, siempre y cuando la parte seleccionada por la primera de las lenses coincida con el todo determinado por la segunda de ellas. Con esto es posible formar lenses que apunten a partes profundas de la estructura de datos inmutables. Tal y como se argumenta en [105], el hecho de que se requiera una invocación a `get` para implementar el método `set` en la lens resultante es un claro indicador de que esta representación podría no ser la más adecuada en términos de composición. Existe un elemento que es neutro para esta función de composición:

```
def id[S, T]: Lens[S, T, S, T] = Lens(identity, const)
```

Se trata de una lens donde la parte seleccionada es exactamente el todo: la lectura es simplemente la función identidad y la escritura ignora el valor actual del todo y lo reemplaza por otro nuevo.

La representación concreta de lenses, a pesar de sus limitaciones en la composición, ofrece una interfaz muy intuitiva y accesible a cualquier programador y es por ello que se explota en algunas librerías industriales, destacando Monocle [119].

van Laarhoven

La representación *van Laarhoven* surge en [123] aunque el nombre es acuñado por O'Connor en [95], quien también es el responsable de ofrecer la versión poli-

mórfica asociada a esta representación [97]. Antes de mostrar la representación, es conveniente introducir una nueva definición:

Definición 7 (Functor)**.** Un *funtor* (covariante) mapea categorías [4] preservando su estructura. En el paradigma funcional se suele proporcionar una type class que clasifica todos aquellos constructores de tipos para los que se puede implementar el método `map`.

```
trait Functor[F[_]] {
  def map[A, B](f: A => B): F[A] => F[B]
}
```

Las instancias de esta clase deben satisfacer las siguientes leyes, donde se asume una notación postfija para `map` (ver apéndice A):

```
def identityLaw[F[_]: Functor, A](fa: F[A]) =
  fa.map(identity) == fa

def andThenLaw[F[_]: Functor, A, B, C](fa: F[A], f: A => B, g: B => C) =
  fa.map(f andThen g) == fa.map(f).map(g)
```

En esencia, la primera ley preserva la flecha identidad, mientras que la segunda ley preserva la composición de flechas.

Existen multitud de constructores que pueden instanciar la clase de funtores. El primero que se mostrará es el de la identidad, representado con la type lambda `λ[x => x]`, que simplemente devuelve el parámetro tipo recibido como entrada:

```
implicit object IdFunctor extends Functor[λ[x => x]] {
  def map[A, B](f: A => B): A => B = f
}
```

En este contexto la implementación de `map` resulta trivial, ya que simplemente devuelve el propio parámetro. Otro funtor que será útil más adelante es el constante, representado como `λ[x => C]`, donde se ignora el tipo de entrada y siempre se devuelve un tipo constante `C`:

```
implicit def constFunctor[C] = new Functor[λ[x => C]] {
  def map[A, B](f: A => B): C => C = identity
}
```

En este caso la función de entrada es ignorada por completo, ya que nunca llegan a instanciarse valores de tipo `A` o `B`. De hecho, el único papel que juegan dichos tipos en este contexto es el de *phantom types* [32]. La implementación de `map` simplemente devuelve la función identidad, que hace que el valor constante pase tal cual. Una vez introducidos los antecedentes necesarios, ya es posible mostrar la nueva representación para lens.

Definición 8 (Up Star)**.** Una función *up star* es aquella que tiene la siguiente estructura:

```
type UpStar[F[_], A, B] = A => F[B]
```

El nombre surge a raíz de la notación matemática $A \to B^*$, donde se pretende indicar que B viene acompañado por cierta estructura adicional, reflejada aquí con el constructor de tipos `F`.

Definición 9 (van Laarhoven Lens)**.** Una lens *van Laarhoven* consiste en una función polimórfica en la que una función *up star* que va de la parte a la parte actualizada se traduce en otra que va del todo al todo actualizado.

```
trait LensV[S, T, A, B] {
  def apply[F[_]: Functor](f: A => F[B]): S => F[T]
}
```

Como se puede apreciar, para que esta definición se corresponda con una lens, es necesario que `F` sea un funtor (definición 7).

El hecho de que esta definición se corresponda con una lens podría no resultar intuitivo a primera vista. Sin embargo, esta abstracción se puede entender como complementaria a la representación concreta, tal y como se explica a continuación. Como se puede apreciar, es necesario transformar una función `A => F[B]` y un valor de tipo `S` en un valor de tipo `F[T]`. La única vía para poder invocar a la función es la de traducir el valor de tipo `S` en un valor de tipo `A`, es decir, la instancia de la lens bajo esta representación tiene que encapsular de alguna manera la función de lectura (`get`). Con esa traducción, sería posible producir un valor de tipo `F[B]`, más cercano al objetivo final (`F[T]`). Sabiendo que `F[_]` es un funtor, simplemente se requeriría una función de tipo `B => T` para abordar ese último paso. Teniendo en cuenta que existe un valor de `S` disponible, es posible determinar que una instancia de lens bajo esta representación también tiene que encapsular el método de actualización (`set`) internamente. Se puede encontrar una caracterización más formal de las conexiones existentes entre la representación concreta y la representación van Laarhoven en [91], donde se explota el lema de Yoneda.

El hecho de que `F[_]` pueda ser cualquier funtor permite obtener comportamientos muy diversos en la invocación de esta función. En este sentido, es posible recuperar `get` mediante el funtor constante:

```
def get[S, T, A, B](ln: LensV[S, T, A, B]): S => A =
  ln[λ[x => A]](identity)
```

La idea reside en recoger el valor de la parte mediante la función `identity` y asumir que se mantendrá constante en el mapeo final mencionado en el párrafo anterior. La implementación de `set` utiliza el funtor identidad:

```
def set[S, T, A, B](ln: LensV[S, T, A, B]): B => S => T = { b =>
  ln[λ[x => x]](const(b))
}
```

Con él se consigue la traducción de una función sobre la parte en una función sobre el todo. La función sobre la parte ignora el valor actual y simplemente devuelve `b`, que se corresponde con el nuevo valor para la parte.

La composición de dos lenses bajo esta representación se muestra a continuación:

```
def andThen[S, T, A, B, C, D](
    ln1: LensV[S, T, A, B],
    ln2: LensV[A, B, C, D]) = new LensV[S, T, C, D] {
  def apply[F[_]: Functor](f: C => F[D]): S => F[T] = ln1(ln2(f))
}
```

Como se puede apreciar, la implementación es muy elegante, donde simplemente se componen las funciones *up star* asociadas a cada instancia. Como consecuencia, el elemento neutro para esta composición se implementa mediante la función identidad:

```
def id[S, T] = new LensV[S, T, S, T] {
  def apply[F[_]: Functor](f: S => F[T]): S => F[T] = f
}
```

La representación van Laarhoven se popularizó principalmente gracias a la librería *lens* [75] de Haskell, aunque también se muestran pinceladas en otras librerías como Monocle [119][9]. La principal ventaja que deriva de ella es la posibilidad de componer ópticas utilizando composición de funciones, en un estilo que curiosamente también resulta natural desde la perspectiva de un programador orientado a objetos [27]. La definición 3 es muy similar a la presentada aquí, siendo la principal diferencia la restricción `Applicative` en vez de `Functor`. Con un funtor aplicativo [89] se consigue la potencia necesaria para poder seleccionar una secuencia de partes [95].

State Monad Morphism

Existen ciertas definiciones que es conveniente introducir antes de poder abordar esta representación, que se presentan a continuación.

Definición 10 (Natural Transformation)**.** Dados un par de funtores `F` y `G` entre las categorías C y D, una transformación natural es una familia de morfismos que transforman objectos sobre `F` en objetos sobre `G`. En Scala, se puede representar como una función polimórfica:

```
trait Natural[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}
```

La transformación debe preservar la siguiente propiedad de conmutación, donde se asumirá `F ⇝ G` como alias para `Natural[F, G]`.

```
def commutes[F[_]: Functor, G[_]: Functor, A, B](
    nat: F ⇝ G,
    fa: F[A],
    f: A => B) =
  nat(fa.map(f)) == nat(fa).map(f)
```

La ley indica que mapear una función sobre el funtor origen y después transformarlo al funtor destino es equivalente a primero transformar al funtor destino y después mapear la función sobre el resultado. Esto permite inferir que una transformación natural cambia la estructura pero en cierto modo preserva el contenido.

La literatura ofrece diferentes maneras de componer transformaciones naturales [4]. Este trabajo sólo manifiesta interés por la composición vertical, que se representa de la siguiente manera.

```
def andThen[F[_], G[_], H[_]](nat1: F ⇝ G, nat2: G ⇝ H): F ⇝ H =
  new Natural[F, H] {
    def apply[A](fa: F[A]): H[A] = nat2(nat1(fa))
  }
```

En esencia, la composición de transformaciones naturales se corresponde con la composición de las funciones polimórficas que se utilizan para representarlas.

Una de las abstracciones más extendidas (y más temidas[10]) en la comunidad de programación funcional es la de mónada. Su popularidad reside en su gran utilidad para representar efectos [127] en este paradigma.

[9] Esta librería utiliza en realidad una representación híbrida entre concreta y van Laarhoven.

[10] `https://two-wrongs.com/the-what-are-monads-fallacy`

Definición 11 (Monad)**.** Una *mónada* es una abstracción que extiende un funtor con una operación `point`, que introduce un valor dentro de un efecto, y una operación `flatMap`, que mapea una función *kleisli* sobre un programa monádico y aplana el resultado.

```
trait Monad[M[_]] extends Functor[M] {
  def point[A](a: A): M[A]
  def flatMap[A, B](f: A => M[B]): M[A] => M[B]
}
```

Las leyes que toda mónada debe cumplir son las siguientes, donde se asume notación postfija para `flatMap`:

```
def leftId[M[_]: Monad, A, B](a: A, f: A => M[B]) =
  point(a).flatMap(f) == f(a)

def rightId[M[_]: Monad, A](ma: M[A]) =
  ma.flatMap(a => point(a)) == ma

def assoc[M[_]: Monad, A, B, C](ma: M[A], f: A => M[B], g: B => M[C]) =
  ma.flatMap(f).flatMap(g) == ma.flatMap(a => f(a).flatMap(g))
```

Básicamente, estas leyes manifiestan que el efecto introducido por `point` es inocuo y que `flatMap` debe de ser asociativo.

Existe una categoría en la que los objetos son mónadas y los morfismos entre dichos objetos son transformaciones naturales que deben preservar ciertas propiedades. Se precisa este tipo de morfismos en la siguiente definición:

Definición 12 (Monad Morphism)**.** Un morfismo de mónadas es una transformación natural entre funtores monádicos `F` y `G` que debe preservar las siguientes leyes.

```
def preserveId[F[_]: Monad, G[_]: Monad, A](morph: F ⇝ G, a: A) =
  morph(point(a)) == point(a)

def preserveFlatmap[F[_]: Monad, G[_]: Monad, A, B](
    morph: F ⇝ G,
    ma: F[A],
    f: A => F[B]) =
  morph(ma.flatMap(f)) == morph(ma).flatMap(a => morph(f(a)))
```

A grandes rasgos, estas leyes mapean un programa sin efectos sobre la mónada original en el correspondiente programa sin efectos sobre la mónada destino y preservan la distributividad sobre `flatMap`.

Como última abstracción antes de mostrar la definición alternativa para una lens, se introduce la mónada *estado*.

Definición 13 (State Monad)**.** *State* es un tipo de datos que se compone de una función que transforma un valor en una nueva versión del mismo, mientras proporciona una salida adicional junto con el valor transformado.

```
case class State[S, A](run: S => (A, S)) {
  def eval(s: S): A = run(s)._1
  def exec(s: S): S = run(s)._2
}
```

La definición incluye `eval` y `exec`, utilidades que se centran en recoger el valor de salida o el estado resultante, respectivamente. Este tipo de datos es una instancia de `Monad`, tal y como se muestra en la figura 2.7.

```
implicit def stateMonad[S] = new Monad[State[S, ?]] {

  def point[A](a: A): State[S, A] = State(s => (a, s))

  def map[A, B](f: A => B): State[S, A] => State[S, B] = { sa =>
    State { s =>
      val (a, s2) = sa.run(s)
      (f(a), s2)
    }
  }

  def flatMap[A, B](f: A => State[S, B]): State[S, A] => State[S, B] = { sa =>
    State { s =>
      val (a, s2) = sa.run(s)
      f(a).run(s2)
    }
  }
}
```

Figura 2.7: Instancia de `Monad` para `State`.

Existen ciertas instancias de la mónada estado que suelen ser muy recurrentes, donde destacan `get` y `set`:

```
object State {
  def get[S]: State[S, S] = State(s => (s, s))
  def set[S](s: S): State[S, Unit] = State(_ => ((), s))
}
```

El primero de los métodos construye una computación que permite extraer el valor actual del estado sin aplicar ninguna modificación sobre éste. El segundo método reemplaza el valor del estado con otro nuevo que se recibe como parámetro.

Finalmente, llegamos a la definición alternativa para lens, que fue introducida por Shkaravska en [111].

Definición 14 (State Monad Morphism Lens)**.** Un morfismo de mónadas de estado es isomorfo a una very well-behaved lens.

```
type LensS[S, A] = State[A, ?] ⇝ State[S, ?]
```

Informalmente, esta definición muestra la traducción de un programa de transformación sobre la parte en una programa de transformación sobre el todo. La principal ganancia que se obtiene de esta representación es que la lens queda definida en términos de un mórfismo, por lo que su composición se reduce a la composición de transformaciones naturales.

Para recuperar la interfaz de la representación concreta es necesario pasar al morfismo de mónada estado el programa de entrada adecuado:

```
def get[S, A](ln: LensS[S, A]): S => A = ln(State.get).eval
def set[S, A](ln: LensS[S, A]): A => S => S = a => ln(State.set(a)).exec
```

Como se puede observar, se utilizan los programas predefinidos `State.get` y `State.set` precisamente para esta tarea y después se utilizan `eval` y `exec` para seleccionar la información de interés para cada caso.

Se desconoce la existencia de una representación basada en transformaciones naturales para el resto de ópticas y consecuentemente no existen librerías de

ópticas que se apoyen en ella. También se desconoce la variante polimórfica asociada a esta representación.

Observación 10. Intuitivamente, la versión indexada de la mónada estado:

```
case class IxState[S, T, A](run: S => (A, T))
```

parecería una firme candidata para adaptar esta representación a su variante polimórfica. Lamentablemente, y aunque es posible traducir una lens concreta en una lens representada mediante `IxState[A, B, ?]` $\rightsquigarrow$ `IxState[S, T, ?]`, no es posible implementar una función inversa que complete el isomorfismo.

Costate comonad coalgebra

Existen varias definiciones que son necesarias para entender esta representación basada en *costate (o store) comonad coalgebra*. A pesar de la complejidad del nombre[11], se mostrará que esta representación es muy similar a la concreta, aunque ofrece conexiones muy interesantes.

Definición 15 (Comonad)**.** Una *comónada* es una abstracción dual a la de una mónada (definición 11), y por tanto se forman invirtiendo las flechas que forman parte de su definición, resultando en `extract` (dual a `point`) y `extend` (dual a `flatMap`):

```
trait Comonad[W[_]] extends Functor[W] {
  def extract[A](wa: W[A]): A
  def extend[A, B](f: W[A] => B): W[A] => W[B]
  def duplicate[A](wa: W[A]): W[W[A]] = cobind[A, W[A]](identity)(wa)
}
```

Se utiliza `W` como nombre para el constructor de tipos ya que visualmente se corresponde con una `M` (de mónada) invertida. La implementación también incluye el método derivado `duplicate` (dual a `join`). Una comónada debe satisfacer las siguientes leyes:

```
def rightId[W[_]: Comonad, A](wa: W[A]) =
  wa.extend(wa2 => extract(wa2)) == wa

def leftId[W[_]: Comonad, A, B](wa: W[A], f: W[A] => B) =
  extract(wa.extend(f)) == f(wa)

def assoc[W[_]: Comonad, A, B, C](wa: W[A], f: W[A] => B, g: W[B] => C) =
  wa.extend(f).extend(g) == wa.extend(wa => g(wa.extend(f)))
```

Como se puede observar, las leyes son también duales a las que pueden encontrarse asociadas a la definición de una mónada.

Definición 16 (F-coalgebra)**.** Una *F-coalgebra* es una función cuya definición gira en torno a un funtor y que tiene la siguiente estructura:

```
type Coalgebra[S, F[_]] = S => F[S]
```

Es posible relacionar los métodos que identifican a una comónada y a una coalgebra, derivando en la siguiente definición.

Definición 17 (Comonad coalgebra)**.** Una *comonad coalgebra* es una coalgebra que cumple con las siguientes leyes:

[11] Aspecto que derivó en más de una broma entre la comunidad de programadores (`https://twitter.com/plt_borat/status/228009057670291456?lang=es`).

```
def idLaw[W[_]: Comonad, S](coalg: Coalgebra[S, W], s: S) =
  extract(coalg(s)) == s

def dupLaw[W[_]: Comonad, S](coalg: Coalgebra[S, W], s: S) =
  coalg(s).map(coalg) == coalg(s).duplicate
```

Es decir, el método `extract` revierte el efecto introducido por `coalg` y el método `duplicate` guarda una correspondencia directa con `coalg`.

Existe una estructura de datos que es dual a la mónada estado (ambas surgen del ajunto formado entre los funtores `S => ?` y `(S, ?)`), que se conoce como la *comónada coestado* (o *store*).

Definición 18 (Store Comonad)**.** *Store* es un tipo de datos que mantiene una posición y un generador de valores a partir de una posición.

```
case class Store[S, A](pos: S, pick: S => A)
```

Este tipo de datos es una instancia de comónada, tal y como evidencia la figura 2.8.

```
implicit def storeComonad[S] = new Comonad[Store[S, ?]] {

  def map[A, B](f: A => B): Store[S, A] => Store[S, B] = { sa =>
    Store(sa.pos, sa.pick andThen f)
  }

  def extract[A](st: Store[S, A]): A = st.pick(st.pos)

  def cobind[A, B](f: Store[S, A] => B): Store[S, A] => Store[S, B] = { sa =>
    Store(sa.pos, s => f(Store(s, sa.pick)))
  }
}
```

Figura 2.8: Instancia de `Store` para `Comonad`

Russell O'Connor ofrece una definición informal muy intuitiva de `Store`[12], donde se define este tipo de datos como un gran almacén de objetos en el que cada uno ocupa una posición. Para acceder a ellos es necesario contar con una carretilla elevadora, donde `pos` determina el lugar en el que ésta se encuentra aparcada. Finalmente llegamos a una nueva definición de lens, donde se utilizan las abstracciones definidas previamente.

Definición 19 (Store comonad coalgebra)**.** Una comonad coalgebra en la que se utilice Store como funtor se corresponde con una representación monomórfica para una lens:

```
case class LensCo[S, A](co: Coalgebra[S, Store[A, ?]])
```

Resulta sorprendente comprobar que las leyes de la comonad coalgebra en este contexto son exactamente las leyes que identifican a una very well-behaved lens.

Si se expande la definición de `Coalgebra` y `Store` en `LensCo` se obtiene la siguiente signatura: `S => (A, A => S)`. Esto es esencialmente la fusión de `get`

[12]La definición se muestra como parte de una respuesta en *StackOverflow*: `https://stackoverflow.com/questions/8766246/what-is-the-store-comonad`

y `set` (reordenando los parámetros de ésta última) en una única función. Por tanto, la recuperación de estos métodos a partir de la nueva representación resulta trivial:

```
def get[S, A](ln: LensCo[S, A]): S => A = s => ln.co(s).pos

def set[S, A](ln: LensCo[S, A]): A => S => S = a => s => ln.co(s).pick(a)
```

La función que permite la composición de lenses bajo esta representación es la siguiente:

```
def andThen[S, A, B](ln1: LensCo[S, A], ln2: LensCo[A, B]): LensCo[S, B] =
  LensCo(s => Store(
    ln2.co(ln1.co(s).pos).pos,
    b => ln1.co(s).pick(ln2.co(ln1.co(s).pos).pick(b))))
```

Como se puede apreciar, tiene atribuidos los mismos problemas que la versión concreta, ya que la composición resulta bastante compleja. La lens identidad se muestra a continuación:

```
def id[S]: LensCo[S, S] = LensCo(s => Store(s, identity))
```

Como se puede apreciar, la variante que se ha mostrado para esta representación ha sido la monomórfica. No obstante, es posible utilizar la noción de store indexado:

```
case class IxStore[A, B, T](pos: A, pick: B => T)
```

para llegar a la variante polimórfica:

```
case class LensCoP[S, T, A, B](co: S => IxStore[A, B, T])
```

Merece la pena destacar que esta definición no se corresponde estrictamente con una F-coalgebra, ya que no respeta la estructura de la definición 16. De hecho, se conoce a esta representación como *indexed costate comonad coalgebroid*. Se desconoce la existencia de librerías que exploten esta representación en su totalidad, ya que resulta menos intuitiva que la representación concreta y tiene atribuidos los mismos problemas de composición. No obstante, esta representación sí que ofrece soporte para codificar traversals, aspecto que se ha explotado para diseñar un constructor que permita crear instancias de esta óptica de manera más sencilla en Monocle[13], contribución derivada de este trabajo.

Profunctor

Como viene siendo habitual a lo largo de esta sección, antes de llegar a la representación de lens basada en profuntores [105], se mostrarán los antecedentes en los que se sustenta.

Definición 20 (Contravariant)**.** Un funtor contravariante es un funtor (definición 7) que invierte el orden de las flechas durante el mapeo entre categorías. En Scala, se representa con una type class cuyo método abstracto es muy similar a `map`, aunque la función de entrada aparece en el orden inverso.

```
trait Contravariant[F[_]] {
  def contramap[A, B](f: B => A): F[A] => F[B]
}
```

[13] https://github.com/julien-truffaut/Monocle/pull/502

De nuevo enfatizamos que `contramap` recibe una función de tipo `B => A` como entrada, en vez de una función de tipo `A => B` como en el caso del funtor. Las leyes asociadas a un funtor contravariante son también duales a las que se asocian a los funtores.

```
def identityLaw[F[_]: Contravariant, A](fa: F[A]) =
  fa.contramap[A](identity) == fa

def andThenLaw[F[_]: Contravariant, A, B, C](fa: F[A], f: B => A, g: C => B) =
  fa.contramap(g andThen f) == fa.contramap(f).contramap(g)
```

Estas leyes preservan la identidad y la composición *inversa* de las funciones.

Definición 21 (Profunctor)**.** Un profuntor es un tipo de funtor, que recibe dos parámetros tipo, y que es contravariante en el primero de ellos y covariante en el segundo. Están definidos en términos de `dimap`, que unifica el mapeo de las posiciones contravariante y covariante.

```
trait Profunctor[P[_, _]] {
  def dimap[A, B, C, D](f: A => B, g: C => D): P[B, C] => P[A, D]
}
```

Un profuntor debe obedecer las siguientes leyes, que no son más que la extensión de las asociadas a los funtores covariantes y contravariantes.

```
def idLaw[P[_, _]: Profunctor, A, B](pab: P[A, B]) =
  pab.dimap[A, B](identity, identity) == pab

def compLaw[P[_, _]: Profunctor, A2, A1, A, B, B1, B2](
    pab: P[A, B],
    f: A2 => A1, g: A1 => A,
    h: B => B1, i: B1 => B2) =
  pab.dimap(g, h).dimap(f, i) == pab.dimap(f andThen g, h andThen i)
```

Estas leyes recogen la preservación de los morfismos de identidad y la preservación de la composición de morfismos, para ambas posiciones.

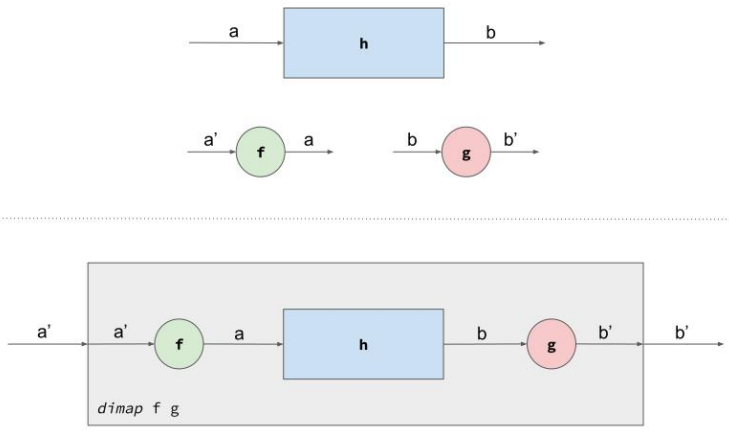

Figura 2.9: Visualización de `dimap` como transformador de cajas

Los profuntores se pueden entender más fácilmente como generalizaciones de funciones. Como consecuencia, podemos verlos como cajas que toman una entrada y producen una salida. El objetivo de `dimap` es por tanto el de adaptar la entrada y la salida de una caja, produciendo una nueva versión de la misma. Bajo esta motivación, se propuso una notación basada en diagramas con el fin de acercar esta abstracción a una audiencia más amplia, que fue utilizada en una serie de posts [84]. La figura 2.9 muestra el efecto que produce `dimap` sobre una caja existente. En la parte superior de la imagen aparecen los componentes de los que parte `dimap`: una caja `h` de tipo `P[A, B]` y morfismos `f` y `g`, representados como círculos. En la parte inferior de la imagen se puede observar la nueva caja resultante de aplicar el transformador `dimap`, donde `f` y `g` se utilizan para adaptar la entrada y la salida de `h`, respectivamente. Las figuras 2.10 y 2.11 muestran la conveniencia de estos diagramas para representar las leyes visualmente.

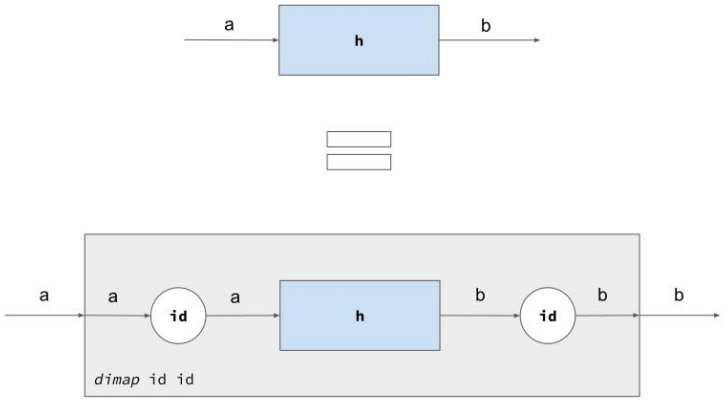

Figura 2.10: Visualización de la propiedad `idLaw`

Existe una familia de profuntores a disposición del programador, donde podemos encontrar `Cartesian`. Dicha abstracción extiende la funcionalidad de un profuntor de la siguiente manera.

Definición 22 (Cartesian Profunctor)**.** Un profuntor cartesiano extiende la entrada y la salida de un componente con cierta información residual. Se codifica mediante la siguiente type class:

```
trait Cartesian[P[_, _]] extends Profunctor[P] {
  def first[A, B, C](h: P[A, B]): P[(A, C), (B, C)]
  def second[A, B, C](h: P[A, B]): P[(C, A), (C, B)]
}
```

Debe satisfacer las siguientes propiedades (junto con sus análogas para `second`, que no se muestran por resultar redundantes):

```
def unitLaw[P[_, _]: Cartesian, A, B](h: P[A, B]) =
  h.dimap(r1, r1p) == first(h)

def assocLaw[P[_, _]: Cartesian, A, B, C, D](h: P[A, B]) =
  first(first(h)).dimap(assoc, assocp) == first[P, A, B, (C, D)](h)
```

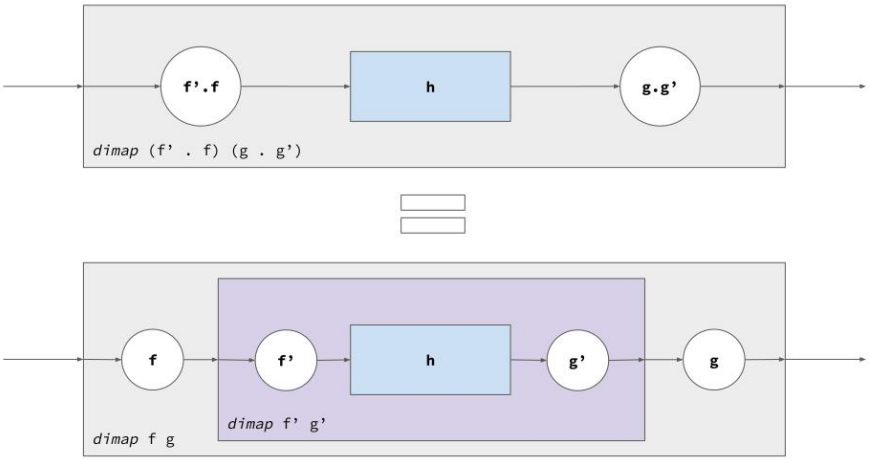

Figura 2.11: Visualización de la propiedad `compLaw`

Estas leyes se apoyan en ciertas funciones auxiliares triviales, cuyas signaturas se presentan a continuación, y cuya implementación no se incluye por ser trivial:

```
def r1[A]: ((A, Unit)) => A
def r1p[A]: A => (A, Unit)
def assoc[A, B, C]: ((A, (B, C))) => ((A, B), C)
def assocp[A, B, C]: (((A, B), C)) => (A, (B, C))
```

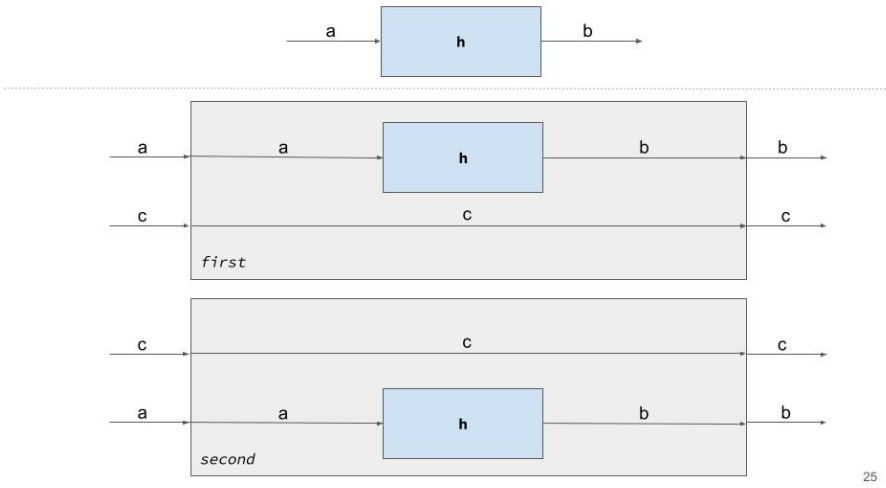

Figura 2.12: Visualización de `Cartesian` como transformador de cajas

De nuevo, es posible utilizar la representación de cajas para ilustrar la transformación que se introduce con `first` y `second` (Figura 2.12). Como se puede apreciar, la única misión de estos métodos es la de extender la entrada y la

salida de `h` con un residuo de tipo `C`, que acompaña a la entrada y salida original, y que pasa a través de la caja resultante tal cual. La diferencia entre `first` y `second` reside en la posición en la que se coloca el residuo. La Figura 2.13 muestra visualmente la propiedad `idLaw`, donde una adaptación por medio de `dimap` que descarta el residuo de la entrada y lo reengancha artificialmente en la salida, debería ser equivalente al efecto producido por `first`. Por su parte, la Figura 2.14 se asocia con la propiedad `assocLaw`, donde se muestra que el anidamiento de residuos por medio de `first` no debería de introducir ningún ruido adicional.

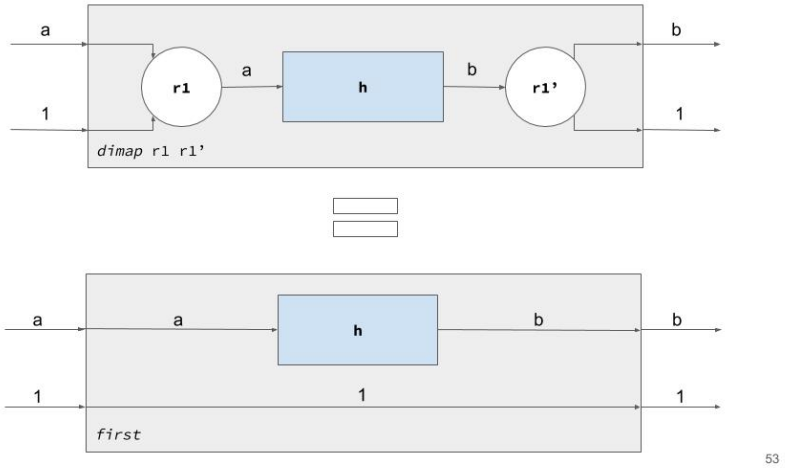

Figura 2.13: Visualización de la propiedad `unitLaw`

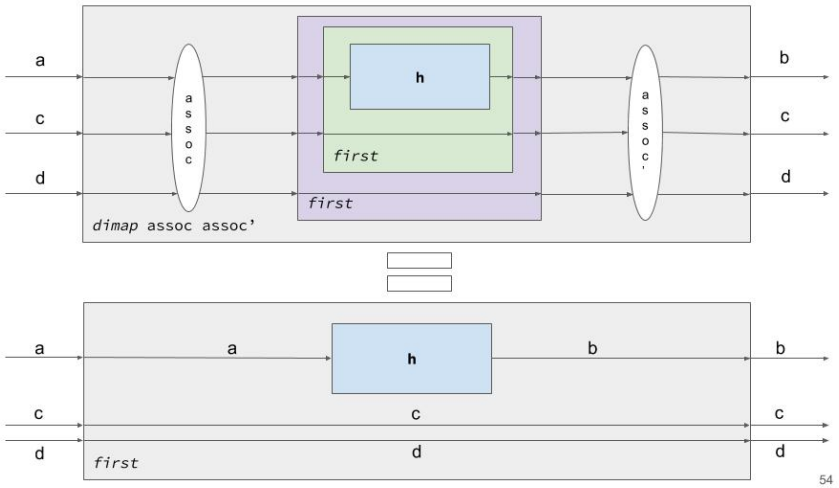

Figura 2.14: Visualización de la propiedad `assocLaw`

Si los profuntores son generalizaciones de funciones, es razonable que el tipo

```
implicit object FunCartesian extends Cartesian[? => ?] {

  def dimap[A, B, C, D](f: A => B, g: C => D): (B => C) => A => D =
    h => a => g(h(f(a)))

  def first[A, B, C](f: A => B): ((A, C)) => (B, C) = {
    case (a, c) => (f(a), c)
  }

  def second[A, B, C](f: A => B): ((C, A)) => (C, B) = {
    case (c, a) => (c, f(a))
  }
}
```

Figura 2.15: Instancia de `Cartesian` para el tipo función.

```
implicit def upStarCartesian[F[_]: Functor] = new Cartesian[UpStar[F, ?, ?]] {

  def dimap[A, B, C, D](
      f: A => B,
      g: C => D): UpStar[F, B, C] => UpStar[F, A, D] = { ubc => a =>
    ubc(f(a)).map(g)
  }

  def first[A, B, C](uab: UpStar[F, A, B]): UpStar[F, (A, C), (B, C)] = {
    case (a, c) => uab(a).map(b => (b, c))
  }

  def second[A, B, C](uab: UpStar[F, A, B]): UpStar[F, (C, A), (C, B)] = {
    case (c, a) => uab(a).map(b => (c, b))
  }
}
```

Figura 2.16: Instancia de `Cartesian` para `UpStar`.

función (`? => ?`) pueda ser una instancia de `Cartesian`, tal y como se muestra en la figura 2.15. La implementación de esta instancia resulta trivial: el método `dimap` se corresponde con la composición de tres funciones; el método `first` simplemente aplica la función sobre el primer elemento de la tupla de entrada, dejando el segundo componente tal cual. Otra estructura que también encaja como `Cartesian` es `UpStar` (definición 8), cuya instancia se recoge en la figura 2.16. La implementación es ligeramente más compleja que la anterior, ya que hay que lidiar con el funtor `F`, pero es autocontenida.

Finalmente llegamos a la representación de lens basada en profuntores, que se muestra en esta definición:

Definición 23 (Profunctor Lens)**.** Una lens basada en profuntores es una función polimórfica que traduce una función generalizada sobre la parte seleccionada en una función generalizada sobre el todo:

```
trait LensP[S, T, A, B] {
  def apply[P[_, _]: Cartesian](h: P[A, B]): P[S, T]
}
```

Como se puede apreciar, `P` tiene que pertenecer a la clase de profuntores cartesianos.

Con la intuición adquirida a partir de la representación van Laarhoven (definición 9) es posible hacerse una idea de la técnica para recuperar los métodos `get` y `set`. Para esta tarea, se debe proporcionar el profuntor adecuado:

```
def get[S, T, A, B](ln: LensP[S, T, A, B]): S => A =
  ln[UpStar[λ[x => A], ?, ?]](identity)(upStarCartesian[λ[x => A]])

def set[S, T, A, B](ln: LensP[S, T, A, B]): B => S => T = { b =>
  ln[? => ?](const(b))
}
```

En este sentido, `get` se recupera utilizando una función up star que trabaje con el funtor constante. Por su parte, `set` se implementa proporcionando el profuntor función. Ambas implementaciones manifiestan un gran parecido con las implementaciones análogas para la representación van Laarhoven. La figura 2.17 nos muestra cómo podríamos ir en la dirección contraria, es decir, se crea una lens basada en profuntores a partir de `get` y `set`.

Observación 11. Instanciar profunctor lenses utilizando este patrón de forma consistente podría no ser lo más adecuado. De hecho, el tipo de transformación que llevan a cabo `first` y `second` son un claro indicador de que una profunctor lens es en realidad más compatible con la representación basada en un isomorfismo [124] (o *residual lens*). Por ejemplo, la implementación de $\pi 1$, la lens que selecciona el primer componente de una tupla, pasaría a ser un mero alias para `first`. En general, es más razonable emplear una lens residual como punto de partida para crear una profunctor lens.

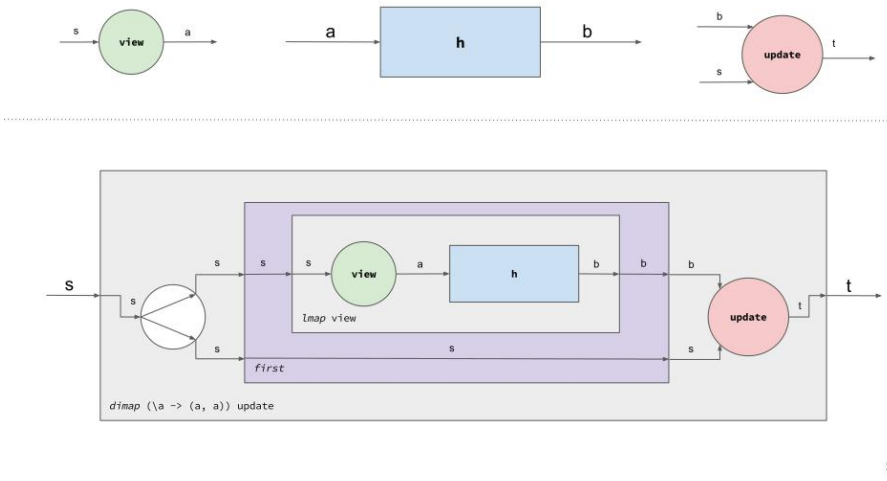

Figura 2.17: Visualización de profunctor lens en términos de `view` y `update`

La composición de lenses se corresponde con la composición de las funciones polimórficas que encapsula la nueva representación:

```
def andThen[S, T, A, B, C, D](
    ln1: LensP[S, T, A, B],
    ln2: LensP[A, B, C, D]) = new LensP[S, T, C, D] {
  def apply[P[_, _]: Profunctor](h: P[C, D]): P[S, T] = ln1(ln2(h))
}
```

El elemento neutro es esencialmente la función identidad, donde la función que actúa sobre la parte es exactamente la función que actúa sobre el todo.

```
def id[S, T] = new LensP[S, T, S, T] {
  def apply[P[_, _]: Profunctor](h: P[S, T]): P[S, T] = h
}
```

A pesar de la complejidad de la definición, existen varias ventajas que han hecho que las profunctor lenses, y más en general, las profunctor optics, hayan captado la atención de gran parte de la comunidad de programadores funcionales. En primer lugar, cada óptica se representa mediante una función, por lo que la composición de las mismas se corresponde con la composición de funciones, lo que resulta ser extremadamente elegante. En segundo lugar, la identificación de nuevas ópticas en la jerarquía pasa por extender o combinar las *constraints* asociadas a las ópticas que ocupan posiciones inferiores en la misma. En lenguajes como Haskell o PureScript, donde estas restricciones se acumulan de forma automática, la composición de funciones también nos sirve para llevar a cabo la composición heterogénea de ópticas. Por último, la codificación de una profunctor lens ofrece una perspectiva muy interesante a la hora de identificar nuevos métodos, jugando con las instancias de `Cartesian`. Por ejemplo, podríamos jugar con *Kleisli* para traducir un efecto computacional sobre el foco en un efector computacional sobre el todo. Por último, es importante destacar que esta representación se explota en librerías como purescript-profunctor-lenses [107], mezzolens [96] y parcialmente en lens [75].

2.1.3. Lenses y efectos

Esta sección describe brevemente trabajo relacionado donde se combinan lenses con computaciones.

Monadic Lens

La literatura muestra varias aproximaciones que pretenden combinar ópticas (más en concreto, lenses), con efectos. La primera de ellas surge en una discusión online [26], donde se propone extender el codominio de los métodos que componen la lens con un efecto monádico. Desafortunadamente, las leyes que parecen naturales para la abstracción resultante restringen sobremanera los efectos en la operación de lectura y parecen dificultar la composición de ésta misma. De aquí deriva una nueva aproximación, las *monadic put-lenses* [103], dónde únicamente es el método de actualización el que extiende su codominio con un efecto monádico. Las leyes asociadas a la nueva aproximación asumen la existencia de una operación de membresía en la mónada que hace que la abstracción no sea general. Finalmente, surge una nueva abstracción [2], que reutiliza los mismos métodos pero ofrece un conjunto alternativo de leyes, y que deriva en la siguiente definición.

Definición 24 (Monadic Lens)**.** Una *monadic lens* extiende la definición de una lens, envolviendo al codominio del método de actualización con un efecto monádico.

```
case class MLens[S, A, M[_]: Monad](mget: S => A, mset: A => S => M[S])
```

Las leyes que definen una *well-behaved* monadic lens se muestran a continuación.

```
def mgetMset[S, A, M[_]: Monad](mln: MLens[S, A, M], s: S) =
  mln.mset(mln.mget(s))(s) == point(s)

def msetMget[S, A, M[_]: Monad, B](
    mln: MLens[S, A, M], s: S, a: A, k: A => S => M[B]) =
  mln.mset(a)(s).flatMap(s2 => k(mln.mget(s2))(s2)) ==
  mln.mset(a)(s).flatMap(k(a))
```

La primera ley (`mgetMset`) indica que no debería de producirse ningún efecto cuando se actualiza el todo con la misma parte contenida por éste. Por su parte, la segunda ley (`msetMget`) impone cierta consistencia entre la lectura de la parte seleccionada y la actualización de la misma. Para recuperar una lens concreta a partir de una monadic lens se debe utilizar la mónada `λ[x => x]` como efecto.

A continuación se muestra uno de los ejemplos más sencillos que se pueden implementar en términos de una monadic lens:

```
def constMLens[A, C](c: C): MLens[A, C, Option] =
  MLens(const(c), c2 => a => if (c != c2) None else Some(a))
```

Esta definición se corresponde con una lens que producirá un fallo cuando se pretenda actualizar la parte seleccionada con un valor distinto a `c`, lo que supone un ejemplo de efecto de parcialidad. Otro ejemplo que caería bajo este paraguas es el de una lens que apunte al valor absoluto de un número entero. Si se pretende actualizar el valor absoluto con un valor negativo, la lens debería producir un fallo, ya que no existe ningún número para el que su valor absoluto sea negativo. Como ejemplo de logging, se podría considerar una lens que registra todos los cambios que ha sufrido la parte seleccionada a lo largo de su vida, donde se tendría que utilizar la mónada `Writer` como efecto monádico.

Observación 12. Las monadic lenses adoptan un repertorio de leyes well-behaved, en lugar de *very well-behaved*, donde una posible ley `msetMset` no se contempla. Dicha ley, al encontrarse con una secuencia de actualizaciones, tendría que descartar los efectos asociados a todas ellas excepto la última. Esta propiedad sería extremadamente restrictiva y limitaría sobremanera el tipo de efectos que se podrían asociar con la lens. De hecho, ninguno de los ejemplos anteriores sería válido si tenemos esta limitación en cuenta. Por ejemplo, en el caso del logging, únicamente se podría registrar el último cambio producido.

Monadic BX

Las transformaciones bidireccionales soportan la evolución de varias fuentes de datos que deben mantener cierta consistencia entre ellas. De hecho, las lenses caen dentro de este paraguas, ya que deben mantener cierta coherencia entre la parte y el todo[14]. En general, es habitual encontrar transformaciones bidireccionales que requieran de efectos adicionales para mantener la consistencia, como situaciones en las que existan varias maneras posibles de mantener la consistencia entre fuentes (no determinismo) o la necesidad de recurrir a servicios externos para completar cierta información requerida (interacción IO).

[14]La lens que hemos introducido en este trabajo se considera una transformación bidireccional asimétrica, ya que su interfaz está orientada a manipular una de las fuentes de datos: el foco. No obstante, también existe una noción de lens simétrica [57], que ofrece funciones para actualizar cada una de las fuentes, que informan también sobre la repercusión de la actualización sobre el resto de fuentes.

Abou-Saleh *et al.* [1] establecen los fundamentos para llevar a cabo transformaciones bidireccionales que requieran de efectos mediante la siguiente estructura de datos.

Definición 25. Una *monadic bx* (mbx) es una transformación bidireccional entre `L` y `R` para la mónada `M`, que ofrece los siguientes métodos.

```
trait BX[M[_], L, R] {
  def getL: M[L]
  def getR: M[R]
  def setL: L => M[Unit]
  def setR: R => M[Unit]
}
```

Una *well-behaved* `BX` es aquella que cumple las siguientes leyes:

```
def getLGetL[M[_]: Monad, L, R](mbx: BX[M, L, R]) =
  mbx.getL.flatMap(l1 => mbx.getL.flatMap(l2 => point((l1, l2)))) ==
  mbx.getL.flatMap(l => point((l, l)))

def getLSetL[M[_]: Monad, L, R](mbx: BX[M, L, R]) =
  mbx.getL.flatMap(mbx.setL) == point(())

def setLSetL[M[_]: Monad, L, R](mbx: BX[M, L, R], l1: L, l2: L) =
  mbx.setL(l1).flatMap(_ => mbx.setL(l2)) == mbx.setL(l2)

/* Definiciones correspondientes para R: getRGetR, getRSetR y setRSetR */
...

def getLgetR[M[_]: Monad, L, R](mbx: BX[M, L, R]) =
  mbx.getL.flatMap(l => mbx.getR.flatMap(r => point((l, r)))) ==
  mbx.getR.flatMap(r => mbx.getL.flatMap(l => point((l, r))))
```

Básicamente, esta definición muestra las leyes de *MonadState* para cada una de las fuentes involucradas, más una ley adicional que establece la conmutatividad de las lectoras `getL` y `getR`. Merece la pena destacar que no tendría sentido incluir una ley que contemple la conmutatividad de las operaciones de actualización, ya que rompería el comportamiento que se espera de una transformación bidireccional.

Teniendo en cuenta que `BX` mantiene dos fuentes de datos en consistencia, sería natural preguntarse si es posible mantener sincronizadas fuentes adicionales respetando las computaciones con efectos. Para esta misión, se propone una noción de composición, cuyos detalles no mostraremos por brevedad[15], donde dadas dos instancias `mbx1: BX[M, A, B]` y `mbx2: BX[M, B, C]`, la composición resultante será de tipo `BX[M, A, C]`. A tener en cuenta, ambas definiciones deben trabajar sobre el mismo tipo de efecto `M` y debe existir una fuente de datos que aparezca en ambas definiciones (en este caso `B`), que actúe como pivote para propagar las actualizaciones entre `A` y `C`.

2.1.4. Composición de ópticas y ejecución de consultas

Una vez introducidas las abstracciones y los combinadores asociados a ellas, es momento de ejercitarlos e ilustrar el estilo y los patrones de diseño propios de

[15] Abou-Saleh *et al.* definen un alias `StateTBX`, que especializa `BX` a instancias de `StateT` (justificando esta decisión con la existencia de un *data refinement* entre las mónadas `M` y `StateT[M, ?, ?]`), sobre el que se define la composición, donde el estado resultante encapsulado por `StateT` es un tipo especial de pares consistentes.

las ópticas. Para esta tarea se han seleccionado tres ejemplos: el primero de ellos es una simplificación del *ejemplo de la universidad* que aparece en la documentación de Monocle[16]; el *ejemplo de las parejas* y el *ejemplo de la organización* han sido extraídos de [19]. Todos ellos servirán para guiar las explicaciones durante las diferentes partes que componen esta tesis.

Ejemplo de la universidad

El primer ejemplo pretende ilustrar el beneficio de las lenses para lidiar con campos anidados en una jerarquía de clases. En particular, se presenta una universidad que alberga un departamento de matemáticas que cuenta con un presupuesto:

```
case class University(unv: String, mathDep: Department)
case class Department(dpt: String, budget: Int)
```

El programador podría estar interesado en duplicar el presupuesto del departamento de matemáticas de la universidad, para lo que podría implementar la siguiente lógica:

```
val doubleUnivBudget: University => University = { u =>
  u.copy(mathDep = u.mathDep.copy(budget = u.mathDep.budget * 2))
}
```

El método `copy` de una case class es muy similar a la técnica de *record syntax* que aparece en lenguajes como Haskell. A pesar de la existencia de esta utilidad, el código resultante no resulta legible y es difícil de mantener, sobre todo si se tiene en cuenta que la lógica que se está implementando es verdaderamente trivial. La situación se vuelve incluso peor cuando se pretende manipular campos que se encuentran a niveles más profundos en la jerarquía.

La popularidad de las lenses (definición 1) reside en que ofrecen una solución muy elegante a este problema, que surge de forma habitual cuando se adoptan estructuras de datos inmutables. Para el ejemplo anterior, se podrían definir diferentes lenses para seleccionar los campos de interés de las diferentes clases. En este caso, se podría seleccionar el departamento de matemáticas de una universidad o el presupuesto de un departamento, que se recogen en el siguiente módulo:

```
object UniversityModel {

  val budgetLn: Lens[Department, Int] =
    Lens(_.budget, b => _.copy(budget = b))

  val mathDepLn: Lens[University, Department] =
    Lens(_.mathDep, md => _.copy(mathDep = md))
}
```

Como se puede apreciar, la estructura de ambas definiciones (donde se explota la notación de *placeholder syntax*) es prácticamente idéntica. Tal es así, que las librerías de ópticas suelen proporcionar utilidades de metaprogramación para generar las lenses asociadas a cada campo de una clase. Si se tiene en cuenta que la parte seleccionada por la primera de las lenses se corresponde con el todo de la segunda, se podrían combinar ambas lenses para producir otra nueva que seleccione el presupuesto del departamento de matemáticas de una universidad:

[16] https://julien-truffaut.github.io/Monocle/examples/university_example.html

```
val univMathBudget: Lens[University, Int] =
  mathDepLn ⋙ budgetLn
```

Esto nos permitiría reimplementar la lógica de `doubleUnivBudget` de una manera más modular:

```
val doubleUnivBudget: University => University =
  univMathBudget ~ (_ * 2)
```

El operador *modify* (∼) es un método derivado de una lens que permite modificar la parte seleccionada aplicando una función sobre ella. Como se puede observar, traduce una lens en una función que trabaja sobre estructuras de datos inmutables. Su definición (en forma de sintaxis infija) es la siguiente:

```
implicit class LensInfix[S, A](ln: Lens[S, A]) {
  def ~(f: A => A): S => S = s => ln.set(f(ln.get(s)))(s)
}
```

La implementación resultante para `doubleUnivBudget` es legible y con regiones muy bien delimitadas: primero se selecciona la parte que resulta de interés; después se decide qué operación se quiere llevar a cabo sobre ella. Estos aspectos aparecían muy acoplados en la versión inicial.

Para mostrar el correcto funcionamiento de la lógica implementada se proporciona la siguiente instancia de la universidad, con un presupuesto más que ajustado:

```
val data: University = University("urjc", Department("math", 500))
```

Aplicando la lógica implementada en `doubleUnivBudget` sobre esta instancia se obtiene el siguiente resultado:

```
val res: University = doubleUnivBudget(data)
// res: University = University("urjc", Department("math", 1000))
```

El comentario en la última línea imprime el valor de `res`, es decir, el resultado de la consulta. Como se puede apreciar, el presupuesto de la universidad se ha doblado, pasando del valor 500 a 1000.

Ejemplo de las parejas

El segundo de los ejemplos presenta una relación sencilla de parejas, donde se proporciona el nombre y la edad de cada persona que la compone[17]. La representación de este modelo con tipos y clases de Scala se muestra en este fragmento de código:

```
type Couples = List[Couple]
case class Couple(her: Person, him: Person)
case class Person(name: String, age: Int)
```

Es importante destacar que estas estructuras de datos están definidas siguiendo un modelo anidado donde las parejas contienen como atributos el propio valor de las personas en vez de claves que permitan recuperar dichos valores. Volveremos a esta caracterización en la sección 6.5, donde la enfrentaremos con la aproximación relacional. Una vez que el modelo queda definido, se proporcionan las ópticas específicas para este ejemplo, aquellas que seleccionan partes relevantes del dominio.

[17] Este ejemplo está obsoleto, especialmente en España, donde el matrimonio entre personas del mismo sexo es legal desde 2005. Sin embargo, hemos decidido mantenerlo tal cual, con el objetivo de facilitar la comparación con respecto a [19], de donde se ha extraído.

```
object CoupleModel {
  val couples: Fold[Couples, Couple] = Fold(identity)
  val her: Getter[Couple, Person] = Getter(_.her)
  val him: Getter[Couple, Person] = Getter(_.him)
  val name: Getter[Person, String] = Getter(_.name)
  val age: Getter[Person, Int] = Getter(_.age)
}
```

De nuevo, se identifican los campos de las clases como partes relevantes a seleccionar. Teniendo en cuenta que los campos son elementos individuales y que la lógica a implementar no contempla actualizaciones, se proponen getters para representar la selección. Adicionalmente, existe un fold `couples` que se utiliza para seleccionar todas las parejas y que utilizaremos como punto de entrada para componer consultas, como se muestra a continuación.

Una vez que se cuenta con los combinadores de ópticas estándar y con las ópticas que caracterizan el dominio de las parejas, es posible producir nuevas ópticas que lleven a cabo selecciones más complejas. Por ejemplo, la siguiente expresión crea una óptica que apunta al nombre y a la diferencia de edad de todas aquellas mujeres que son más mayores que sus parejas:

```
val differencesFl: Fold[Couples, (String, Int)] =
  couples ⋙ filtered[Couple](c => c.her.age > c.him.age) ⋙
    (her ⋙ name) *** ((her ⋙ age) - (him ⋙ age))
```

Tal y como se mencionaba con anterioridad, `couples` se posiciona como un punto de entrada que nos permite empezar a seleccionar información del dominio. Se utiliza `filtered` para eliminar de la selección a todas aquellas parejas donde la edad de la mujer no sea mayor que la del hombre. Finalmente se utiliza `***` para juntar la información que se desea recopilar: el nombre de la mujer y la diferencia de edad.

El siguiente paso es el de transformar la óptica resultante en una consulta que permita extraer la información a partir de una instancia del modelo, es decir, una función que reciba un objeto `Couples` como entrada y devuelva los valores correspondientes. Esta tarea es trivial, simplemente se debe invocar el método `getAll` (definición 6).

```
val differences: Couples => List[(String, Int)] =
  differencesFl.getAll
```

Si se alimenta esta consulta con la misma información con la que se trabajaba en el ejemplo original [19], se debería esperar el mismo resultado:

```
val data: Couples = List(
  Couple(Person("Alex", 60), Person("Bert", 55)),
  Couple(Person("Cora", 33), Person("Drew", 31)),
  Couple(Person("Edna", 21), Person("Fred", 60)))

val res: List[(String, Int)] = differences(data)
// res: List[(String, Int)] = List((Alex,5), (Cora,2))
```

Como era de esperar, indica que Alex y Cora son mayores que sus parejas con una diferencia de edad de 5 y 2 años, respectivamente.

Ejemplo de la organización

El último de los ejemplos que presentamos es el de una organización compuesta por departamentos donde trabajan una serie de empleados. Cada uno de

ellos es capaz de desempeñar un conjunto de tareas. Proponemos el siguiente modelo para recoger esta funcionalidad:

```
type Org = List[Department]
case class Department(dpt: String, employees: List[Employee])
case class Employee(emp: String, tasks: List[Task])
case class Task(tsk: String)
```

Una vez que el modelo está definido, es posible definir el conjunto de ópticas específicas de este dominio:

```
object OrgModel {
  val departments: Fold[Org, Department] = Fold(identity)
  val dpt: Getter[Department, String] = Getter(_.dpt)
  val employees: Fold[Department, Employee] = Fold(_.employees)
  val emp: Getter[Employee, String] = Getter(_.emp)
  val tasks: Fold[Employee, Task] = Fold(_.tasks)
  val tsk: Getter[Task, String] = Getter(_.tsk)
}
```

En este modelo existen ciertos campos que apuntan a una lista de partes, como pueden ser `employees` o `tasks`, por lo que se utilizan folds en lugar de getters, que son abstracciones que permiten lidiar con secuencias más fácilmente. También se despliegan varios getters para campos sencillos y `departments`, cuyo propósito es análogo al de la óptica `couples` que se mostró en el ejemplo anterior. Podemos utilizar estas definiciones y los combinadores estándar para componer una óptica que seleccione el nombre de aquellos departamentos en los que todos sus trabajadores sepan realizar la tarea de *abstraer*:

```
val expertiseFl: Fold[Org, String] =
  departments ⋙ filtered(
    _.employees.forall(_.tasks.map(_.tsk).contains("abstract"))) ⋙ dpt
```

Esta expresión se apoya en `departments` para empezar a seleccionar información, filtra los departamentos donde todos los empleados contienen `"abstract"` como parte de sus habilidades y finalmente selecciona el nombre del departamento.

Observación 13. Como se puede apreciar, `filtered` recibe una función como predicado. Este predicado en particular requiere acceder a ciertos niveles de profundidad para determinar si todos los empleados del departamento saben abstraer. A pesar de que las ópticas son idóneas para este tipo de situaciones, el predicado prescinde ellas, utilizando en su lugar la interfaz de colecciones de Scala. Esto deriva en una implementación híbrida entre ópticas y colecciones que resulta confusa y poco natural. La siguiente versión, aún no siendo idiomática, explota mejor el potencial de las ópticas y hace que la implementación sea más homogénea:

```
val expertiseFl: Fold[Org, String] =
  departments ⋙ filtered(
    employees.all((tasks ⋙ tsk).elem("abstract").get).get) ⋙ dpt
```

Es importante destacar que `employees`, `tasks` y `tsk` se corresponden con las ópticas de `OrgModel` y no directamente con los propios campos, como ocurría en la versión anterior. El capítulo 6 propone una refactorización de los combinadores para hacer que este estilo se utilice por defecto.

De nuevo, traducir la óptica en una consulta resulta ser una tarea trivial, en la que simplemente es necesario invocar `getAll`:

```
def expertise: Org => List[String] =
  expertiseFl.getAll
```

Ya es posible invocar esta función pasando una organización como parámetro, donde se vuelve a recurrir a los datos extraídos del ejemplo original:

```
val data: Org = List(
  Department("Product", List(
    Employee("Alex", List(Task("build"))),
    Employee("Bert", List(Task("build"))))),
  Department("Quality", List.empty),
  Department("Research", List(
    Employee("Cora", List(Task("abstract"), Task("build"), Task("design"))),
    Employee("Drew", List(Task("abstract"), Task("design"))),
    Employee("Edna", List(Task("abstract"), Task("call"), Task("design"))))),
  Department("Sales", List(
    Employee("Fred", List(Task("call"))))))

val res: List[String] = expertise(data)
// res: List[String] = List(Quality, Research)
```

El resultado de aplicar este estado a la consulta nos indica que los departamentos de calidad e investigación son los únicos en los que todos los empleados saben llevar a cabo la tarea de abstraer.

Observación 14. Nos referimos a `get`, `getAll`, etc. como *consultas* derivadas de las ópticas. En particular, las ópticas de sólo lectura simplemente producen consultas sencillas de lectura, en una correspondencia de uno a uno. Como consecuencia, la separación de aspectos entre expresiones de ópticas y consultas derivadas de ellas no es tan clara como en otras ópticas, como por ejemplo las lenses, que cuentan en su catálogo con consultas para leer (`get`), actualizar (`set`) y reemplazar ($\sim$) la parte que seleccionan. Discutiremos las implicaciones de este aspecto en el capítulo 8.3.

2.2. Repositorios

El patrón *repository* [34] es un patrón muy extendido en la industria del software, cuya principal misión es la de abstraer los detalles de la capa de datos. A grandes rasgos se trata de una interfaz que alberga primitivas para recuperar y reemplazar el valor de una determinada entidad. De esta manera, se podría considerar el siguiente repositorio para acceder a la información de una universidad muy sencilla:

```
trait UniversityRepo {
  def getName: String
  def setName(s: String): Unit
  def modName(f: String => String): Unit
  def getDepartments: List[Department]
  def setDepartments(ds: List[Department]): Unit
  def modDepartments(f: Department => Department): Unit
}
```

Las interfaces pueden representarse en Scala mediante un *trait* (ver apéndice A). Como se puede apreciar, la interfaz contiene primitivas para acceder, reemplazar y modificar el nombre y la lista de departamentos de la universidad. Merece la pena destacar que *Unit* se corresponde con la tupla vacía `()`, es decir, es el tipo con una única instancia que puede utilizarse cuando no se pretende devolver ninguna información de interés.

La PF ha resultado ser de utilidad en combinación con los patrones y técnicas del campo de DDD [36]. En el caso particular de los repositorios, la principal

ganancia obtenida por adoptar la PF resulta estar en la implementación de la lógica de negocio. No obstante, el documento propone hacer uso de las estructuras de datos funcionales para contemplar la posibilidad de errores en el acceso a los datos. En concreto, se propone actualizar el repositorio de la siguiente manera:

```
trait UniversityRepoErr {
  def getName: Try[String]
  def setName(s: String): Try[Unit]
  def modName(f: String => String): Try[Unit]
  def getDepartments: Try[List[Department]]
  def setDepartments(ds: List[Department]): Try[Unit]
  def modDepartments(f: Department => Department): Try[Unit]
}
```

El único cambio introducido es que los tipos asociados a los diversos resultados se envuelven mediante `Try`, que contempla dos situaciones: o bien contiene un valor del tipo asociado, o bien contiene un error indicando qué es lo que ha ido mal en el acceso. Esta es la solución ofrecida por la PF para lidiar con errores de forma pura, evitando la introducción de indeseadas excepciones, y haciendo partícipe al compilador y al usuario de la interfaz de que algo podría ir mal durante la invocación de la primitiva.

La solución propuesta en `UniversityRepoErr` no resulta ser lo suficientemente general para muchas situaciones. Por ejemplo, se supondrá el caso en el que se pretende dar una instancia del repositorio para lidiar con Slick, que en sus versiones más recientes soporta un acceso asíncrono mediante la introducción de `Future`s. La implementación particular de `getName` podría ser la siguiente:

```
def getName: Try[String] = {
  val res: Future[String] = ... // Slick query to get university's name
  Await.result(res)
}
```

La implementación es correcta, pero la llamada a `Await.result` está rompiendo la asincronía para adaptarse a la signatura estipulada por el repositorio. Por tanto, esta interfaz no es capaz de beneficiarse de las mejoras introducidas en las últimas versiones de Slick. En el caso de que la asincronía fuese imprescindible, habría que modificar las interfaces, lo cual rompería toda la lógica de negocio que accediese a ella, que además quedaría contaminada con los contextos de ejecución necesarios para desarrollar código que contiene `Future`s. La abstracción ofrecida por los repositorios es por tanto *leaky*[113].

Se propone ir un paso más allá para generalizar los repositorios, mediante la introducción de un constructor de tipos abstracto que parametrice la interfaz, con el objetivo de abstraerse sobre el tipo de efecto (error, asincronía, etc.), con lo que se llega al concepto de lo que denominamos *repositorio funcional*, que aplicado al ejemplo de la universidad resulta en la siguiente definición:

```
trait UniversityAlg[P[_]] {
  def getName: P[String]
  def setName(s: String): P[Unit]
  def modName(f: String => String): P[Unit]
  def getDepartments: P[List[Department]]
  def setDepartments(ds: List[Department]): P[Unit]
  def modDepartments(f: Department => Department): P[Unit]
}
```

Esta nueva abstracción define la clase de efectos computacionales `P` que se podría usar para acceder a los datos subyacentes. La lógica de negocio se podría implementar en términos de los métodos que definen esta interfaz, manteniendo al programador al margen de los detalles específicos de la infraestructura. No obstante, al introducir este nivel de abstracción, se requerirían interfaces adicionales para poder componer los programas resultantes y las posibles leyes que deben satisfacer las primitivas, donde las interfaces propuestas por funtores aplicativos [89] o mónadas [127] son idóneas. Con esto se estaría entrando de lleno en el estilo MTL, que se describe en la sección 2.3. En cualquier caso, esta aproximación cuenta con ciertas limitaciones que se detallan a continuación:

- En primer lugar, a pesar de que `UniversityAlg` esconde el estado de la universidad bajo el efecto `P`, algunos de sus métodos exponen referencias a `Department` en sus signaturas. Sin embargo, la información asociada a un departamento podría ser tan grande que hiciera que su instancia en memoria fuera ineficiente o directamente impracticable. En esta circunstancia, se podría contemplar un repositorio o teoría algebraica adicional, desacoplada de `UniversityAlg`, encargada de la evolución de los miembros del departamento a través de su propio efecto `Q`. El problema de esta aproximación es que la conciliación de los programas heterogéneos `P`, `Q`, etc. generados por las diversas teorías introduce una notable complejidad, especialmente cuando el modelo tiene asociados muchos niveles de anidamiento.

- En segundo lugar, no se están utilizando abstracciones estándar para describir los métodos que conforman la interfaz del repositorio. Consecuentemente, `UniversityAlg` contiene métodos de grano muy fino, que tienden a repetirse una y otra vez para cada campo del modelo. En este sentido, sería posible reemplazar `getName` y `modName` con una evidencia de `MonadState` [37, 39], una de las teorías algebraicas más populares en MTL, que ponga foco en el nombre. Esta abstracción permite la visualización y actualización del campo bajo una perspectiva general. Lamentablemente, se desconoce la existencia de teorías algebraicas estándar que ofrezcan interfaces específicas adaptadas para secuencias de valores que permitan contemplar el caso de los departamentos.

- Las tecnologías que almacenan la información de la universidad y la que almacena la información de un departamento podrían ser diferentes o podrían estar distribuidas en diferentes nodos. La instanciación de un `P[_]` que sea capaz de unificar todos estos aspectos podría resultar especialmente complejo o inviable.

2.3. MTL

Esta sección introduce un estilo de programación que ha adquirido una gran notoriedad y multitud de debates en la comunidad de programadores funcionales para lidiar con efectos: estado mutable, control de errores, no determinismo, etc. Aunque el término MTL se establece como las siglas de *monad transformer library*, existe cierta polémica sobre si realmente es una librería de transformadores de mónadas [79], donde la constante refactorización de las librerías

involucradas y las dependencias entre ellas tampoco ha ayudado a clarificar este aspecto [55]. En este sentido, este trabajo recoge en este término tanto a los transformadores como a las teorías algebraicas que ofrecen un acceso uniforme a éstos. En lo siguiente, se describirán las abstracciones más relevantes, con el objetivo de mostrar el estilo que denota la aproximación. Para guiar las explicaciones, se utilizará como ejemplo un lenguaje de operaciones muy sencillo, extraído de [127]. Dicho lenguaje permite formar expresiones a partir de constantes y divisiones, de las que se pretende realizar una evaluación potencialmente *effectful*.

```
sealed abstract class Expr
case class Con(a: Int) extends Expr
case class Div(t: Expr, u: Expr) extends Expr
```

Como se puede apreciar, el lenguaje se presenta como un ADT. Esta codificación también se denomina *inicial*, aspecto que se desarrolla más adelante en la sección 2.4.

2.3.1. Mónadas y programación imperativa

Dada una expresión del nuevo lenguaje, el programador podría estar interesado en llevar a cabo su evaluación. Si se tiene en cuenta que una división podría producir un error cuando el divisor es cero, este aspecto debería quedar reflejado en la signatura del evaluador. Por ello, se envuelve el resultado con el constructor de tipos `Option`, con lo que se llega a esta implementación:

```
def evalOp(t: Expr): Option[Int] = t match {
  case Con(a) => Some(a)
  case Div(u, v) => evalOp(u) match {
    case Some(a) => evalOp(v) match {
      case Some(b) => if (b == 0) None else Some(a / b)
      case None => None
    }
    case None => None
  }
}
```

En el caso de encontrar una constante, simplemente se envuelve el resultado mediante `Some`. En el caso de que aparezca una división se reflejará la situación errónea con un `None`, siempre y cuando el divisor evalue a cero. En caso contrario, se envuelve el resultado de la división con un `Some`. Antes de esto, es necesario contemplar la posibilidad de que las expresiones que forman el dividendo y el divisor también puedan producir una situación excepcional durante su evaluación, y en tal caso se debe propagar el error. En general, el código mostrado es poco legible y difícil de mantener, debido principalmente al acoplamiento existente entre la evaluación y la gestión de errores.

Dejando los errores de lado, se pretende ahora contar el número de divisiones que se han registrado durante la evaluación. De nuevo, la signatura de la función debe reflejar este aspecto, extendiendo el resultado con un acumulador. La función de evaluación resultante podría definirse de la siguiente manera:

```
def evalSt(t: Expr)(acc: Int): (Int, Int) = t match {
  case Con(a) => (acc, a)
  case Div(u, v) => {
    val (a, acc1) = evalSt(u)(acc)
    val (b, acc2) = evalSt(t)(acc1)
```

```
    val acc3 = acc2 + 1
    (acc3, a / b)
  }
}
```

En el caso de encontrarse una constante, se devuelve la evaluación de éste junto con el acumulador actual intacto. Si por el contrario se trata de una división, se debe de contar el número de divisiones que aparecen en el dividendo y en el divisor, y sumar una unidad al acumulador resultante. Este tipo de código es tedioso y muy propenso a errores, ya que es fácil equivocarse en el hilado del acumulador. Además, se trata de una función parcial, ya que se producirá un fallo cuando el divisor evalúe a cero.

Los ejemplos anteriores introducen efectos sobre la evaluación de los términos, el primero mediante la gestión de errores y el segundo mediante la manipulación de un estado. Moggi demostró cómo estos (y otros) efectos pueden modelarse mediante el concepto de *mónada* [92] (definición 11) y Wadler se encargó de popularizar la idea en la comunidad de programación funcional [127], donde suele aparecer en forma de *type class* [129], como ya se mostró en la definición 11.

Es posible instanciar una mónada que contemple la propagación de errores. Para ello, no hay más que dar una instancia de mónada para el constructor `Option`.

```
implicit object optionMonad extends Monad[Option] {
  def point[A](a: A): Option[A] = Some(a)
  def bind[A, B](ma: Option[A])(f: A => Option[B]): Option[B] = ma match {
    case Some(a) => f(a)
    case None => None
  }
}
```

Mientras que `point` simplemente envuelve el valor en un `Some`, `bind` se encarga de propagar los posibles errores. Con esta definición ya se dispone de todo lo necesario para abordar la reimplementación de la función de evaluación con los nuevos combinadores, lo que resultaría en el siguiente código:

```
def evalOp(t: Expr): Option[Int] = t match {
  case Con(a) => point(a)
  case Div(u, v) =>
    evalOp(u) >>= { a =>
      evalOp(v) >>= { b =>
        if (b == 0) None else point(a / b)
      }
    }
}
```

Esta definición es claramente más concisa que la original, aunque su legibilidad podría resultar confusa para un lector que no está acostumbrado al estilo. Afortunadamente, existe una notación inspirada en las comprehensions de teoría de conjuntos que puede resultar muy conveniente para este contexto [126]. Aunque en Haskell se popularizó bajo el nombre de *do-notation*, en Scala adquiere la denominación de *for-comprehension* [98][18]. El estilo también fomenta la introducción de combinadores estándar, con lo que se podría llegar a la siguiente implementación para la función de evaluación:

[18] Existen ciertas diferencias entre una do-notation y una for-comprehension, aunque no son relevantes en el contexto de este trabajo de investigación, y por tanto serán ignoradas.

```
def evalOp(t: Expr): Option[Int] = t match {
  case Con(a) => point(a)
  case Div(u, v) => for {
    a ← evalOp(u)
    b ← evalOp(v)
    _ ← None.whenM(b == 0)
  } yield a / b
}
```

Ahora se aprecia más claramente lo que está ocurriendo en el caso de que nos encontremos con una división: primero se evalúa el dividendo, después se evalúa el divisor y por último se devuelve el resultado de la división. Antes de éste último paso se elevará un error siempre y cuando el dividendo sea evaluado a cero, que se consigue con el combinador `whenM`. Sin embargo, no hay necesidad de inspeccionar los términos de la evaluación ni de propagar los errores, tarea que pasa a ser responsabilidad de los combinadores monádicos de forma transparente.

Se puede repetir el proceso para contemplar el efecto de la manipulación de un estado. Afortunadamente, esta instancia ya se introdujo en la definición 13, necesaria para mostrar una representación particular de lens. Dicha instancia permite reimplementar `evalSt` de la siguiente manera:

```
def evalSt(t: Expr): State[Int, Int] = t match {
  case Con(a) => point(a)
  case Div(u, v) => for {
    a ← evalSt(u)
    b ← evalSt(v)
    _ ← modify(_ + 1)
  } yield a / b
}
```

De nuevo el código resultante es muy legible: se evalúa el dividendo, se evalúa el divisor y finalmente se devuelve el resultado de la división. Justo antes, se actualiza el contador de divisiones utilizando `modify`, que modifica el estado actual utilizando la función pasada como argumento, devolviendo un valor `Unit` como salida (que es ignorado).

Desde una perspectiva más general, y gracias a la sintaxis introducida mediante una for comprehension, es posible entender la interfaz de una mónada como una interfaz para hacer programas imperativos. En este sentido, el método `flatMap` guarda una cierta analogía con el símbolo `;` que suele utilizarse para delimitar instrucciones; de la misma manera, el método `point` guarda una correspondencia con la típica instrucción **`return`**. Estas analogías quedan de manifiesto con una instancia muy particular de mónada, que se presenta a continuación:

```
implicit object idMonad extends Monad[λ[x => x]] {
  def point[A](a: A): A = a
  def bind[A, B](ma: A)(f: A => B): B = f(ma)
}
```

Esta mónada es conocida como la mónada *Id* y se corresponde con una computación libre de efectos. Ahora, un programa sencillo y eminentemente imperativo, como el que se muestra a continuación:

```
val res: Int = {
  val x = 2;
  val y = 3;
  return (x + y);
}
```

podría ser reimplementado en términos de la mónada *Id* de la siguiente manera:

```
val res: Int = for {
  x ← 2
  y ← 3
} yield (x + y)
```

donde puede apreciarse esa noción imperativa en su forma más primitiva.

Llegados a este punto, podría resultar de interés extender el evaluador para que combine diversos tipos de efectos, como por ejemplo, controlando los errores y contando el número de divisiones simultáneamente. Lamentablemente surgen ciertas limitaciones en la composición de mónadas para las que es necesario recurrir a nuevas abstracciones, tal y como se muestra en la próxima sección.

2.3.2. Monad Transformers

En un escenario real, es más que razonable esperar que la función de evaluación pueda requerir la combinación de varios efectos a la vez. Si se pretende mezclar la gestión de errores con la manipulación de un estado, se podría llegar a la siguiente función de evaluación:

```
def eval(t: Expr): State[Int, Option[Int]] = t match {
  case Con(a) => point(point(a))
  case Div(u, v) => for {
    oa ← eval(u)
    r1 ← oa match {
      case Some(a) => for {
        ob ← eval(v)
        r2 ← ob match {
          case Some(b) => {
            if (b == 0) point(None) else modify(_ + 1).as(point(a / b))
          }
          case None => point(None)
        }
      } yield r2
      case None => point(None)
    }
  } yield r1
}
```

El tipo del resultado es `State[Int, Option[Int]]`, con lo que se podría esperar que la función devolviese el resultado opcional de la evaluación acompañado por el número de divisiones existentes, o al menos el número de éstas hasta que se produjo un error. La implementación resulta bastante compleja, recordando a la primera versión de `evalOp`, donde la propagación de errores era completamente manual. De hecho, a duras penas se pueden reutilizar las ventajas de la mónada `Option` en este escenario.

El problema de fondo reside en que a diferencia de los funtores o los funtores aplicativos [89], las mónadas no componen, es decir si `M[_]` y `N[_]` son mónadas, `M[N[_]]` o `N[M[_]]` no necesariamente tienen por qué serlo. Con el objetivo de paliar esta situación, surgen los *transformadores de mónadas* [80], abstracciones componibles que persiguen la combinación de efectos.

Definición 26 (Monad Transformer)**.** Un transformador de mónadas construye una nueva mónada a partir de otra ya existente. Contiene un método `liftM` para adaptar programas sobre la mónada subyacente `M` en programas sobre la nueva mónada `T[M, ?]`.

```
trait MonadTrans[T[_[_], _]] {
  def liftM[M[_]: Monad, A](ma: M[A]): T[M, A]
}
```

Las leyes que debe cumplir todo transformador de monadas son las siguientes:

```
liftM(point(a)) == point(a)
liftM(ma >>= f)  == liftM(ma) >>= (a => liftM(f(a)))
```

Puesto en palabras, `liftM` debe preservar `point` y debe ser distributivo sobre `bind`.

Existe un transformador que nos permite contemplar el efecto introducido por `Option`. Se denomina `OptionT[M[_], A]` y representa una computación de tipo `M[Option[A]]`. Su instancia para `Monad` es la siguiente:

```
implicit def optionTMonad[M[_]: Monad] = new Monad[OptionT[M, ?]] {
  def point[A](a: A): OptionT[M, A] = OptionT(point(point(a)))
  def bind[A, B](ma: OptionT[M, A])(f: A => OptionT[M, B]): OptionT[M, B] =
    OptionT(for {
      oa ← ma.run
      ob ← oa match {
        case Some(a) => f(a).run
        case None => point(None)
      }
    } yield ob)
}
```

La implementación de `point` es trivial, simplemente hay que envolver el valor `a` con las capas pertinentes. Por su parte, el método `bind` se encarga de inspeccionar la salida del programa original, procediendo normalmente si el valor está definido o propagando el error en caso contrario. A continuación se muestra la instancia de `MonadTrans` para este transformador.

```
implicit object optionTMonadTrans extends MonadTrans[OptionT] {
  def liftM[M[_]: Monad, A](ma: M[A]): OptionT[M, A] = OptionT(ma.map(point))
}
```

Como se puede apreciar, la implementación es muy sencilla, el método `liftM` mapea el valor contenido en el programa `ma`, envolviéndolo con un `Some`, o dicho de otra manera, no introduce ningún error durante la adaptación. Ahora sí, ya se dispone de todos los ingredientes para implementar la función de evaluación bajo la nueva abstracción.

```
def eval(t: Expr): OptionT[State[Int, ?], Int] = t match {
  case Con(a) => point(a)
  case Div(u, v) => for {
    a ← eval(u)
    b ← eval(v)
    _ ← if (b == 0) none else liftM(modify(_ + 1))
  } yield a / b
}
```

El código resulta bastante natural y muy similar a los evaluadores que trabajaban con un único efecto de la sección anterior. La mayor diferencia se encuentra en la línea que chequea el divisor. En caso de ser cero, se utiliza `none`, que no es más que un constructor de `OptionT` con el que se introduce una situación errónea. Por otro lado, si el divisor es distinto de cero, se actualiza el contador mediante `modify`, aunque en esta ocasión es necesario que venga acompañado de `liftM`, para traducir el programa de `State` en un programa compatible con el transformador.

Al igual que se ha introducido un transformador para contemplar el efecto de la gestión de errores, también se podría haber introducido un transformador asociado al efecto de la manipulación de estado. Concretamente, se trata de `StateT[S, M, A]`, que alberga un programa de tipo `S => M[(S, A)]`. Como es de esperar, este transformador genera una mónada a partir de otra dada como argumento:

```
implicit def stateTMonad[S, M[_]: Monad] = new Monad[StateT[M, S, ?]] {
  def point[A](a: A): StateT[M, S, A] = StateT(s => point((s, a)))
  def bind[A, B](ma: StateT[M, S, A])(f: A => StateT[M, S, B]): StateT[M, S, B] =
    StateT(s => ma(s) >>= { case (s1, a) => f(a)(s1) })
}
```

Y evidentemente, instancia la clase asociada a un transformador de mónadas.

```
implicit def stateTMonadTrans[S] = new MonadTrans[StateT[?[_], S, ?]] {
  def liftM[M[_]: Monad, A](ma: M[A]): StateT[M, S, A] = StateT(ma.strengthL)
}
```

Esta instancia utiliza `strengthL` para extender el contenido de `ma` con el estado original, manifestando así que no se lleva a cabo ninguna actualización sobre dicho estado, que pasa tal cual. A pesar de la ganancia introducida con las nuevas abstracciones, existen ciertas limitaciones asociadas a ellas:

- El orden en el que se componen los efectos es importante. Por ejemplo, `OptionT[State[S, ?], A]` es diferente a `StateT[Option, S, A]`. De hecho, el primero de ellos induce una computación de tipo `S => (S, Option[A])`, mientras que el segundo produce una función `S => Option[(S, A)]`. Volviendo al ejemplo de las divisiones, la primera versión nos permitiría calcular el número de divisiones que se han encontrado hasta detectar un error, mientras que el segundo sólo devolvería dicho contador en caso de que la evaluación no produjese un error.

- Añadir nuevos efectos o invertir el orden de los mismos rompe la lógica de negocio, ya que las invocaciones a `liftM` deben reflejar el mismo orden que el recogido en la pila de efectos. Por tanto, el código está muy atado a una configuración muy específica y por tanto es poco mantenible. En general, es deseable introducir mecanismos de abstracción que permitan una mejor modularización [61], desacoplando la lógica de las computaciones particulares.

- Lidiar con `liftM` resulta en una tarea tediosa, especialmente cuando se trabaja con pilas de efectos largas y complejas. De hecho, la adaptación de un programa asociado a un efecto al conjunto global que conforma la pila requiere de tantas invocaciones a `liftM` como niveles de anidamiento existen hasta llegar a él.

Por todo ello, la librería MTL viene acompañada por un conjunto de type classes que proporcionan un acceso a los efectos más uniforme. Estas técnicas, que no están exentas de limitaciones, quedan definidos en la próxima sección.

2.3.3. Teorías algebraicas de efectos

Con el objetivo de proveer una interfaz más uniforme para el acceso a las primitivas que introducen los diferentes efectos, e inspirados en [66], MTL despliega

un repertorio de type classes que permiten abstraerse de transformadores particulares. En el contexto de esta tesis, es de especial relevancia la clase asociada a la manipulación de un estado, que se define a continuación.

Definición 27 (MonadState)**.** `MonadState` clasifica todos aquellos efectos que son capaces de manipular y acceder a un estado interno. Proporciona un par de métodos, el primero de los cuales recupera el estado actual y el segundo reemplaza el estado actual por otro nuevo recibido como parámetro.

```
trait MonadState[M[_], S] extends Monad[M] {
  def get: M[S]
  def put(s: S): M[Unit]
}
```

Estas son las leyes que debe cumplir toda instancia de `MonadState` para ser considerada válida:

```
get >>= (s => get >>= (s2 => k(s, s2))) == get >>= (s => k(s, s))
get >>= put == point(())
put s >>= get == put s >>= point(s)
put s1 >> put s2 == put s2
```

La primera ley manifiesta que la recuperación del estado no debería producir ningún efecto adicional inesperado; la segunda indica que reemplazar con el estado actual es equivalente a no hacer nada; la tercera ley establece que la recuperación del estado justo después de su escritura debe devolver el nuevo valor escrito; y por último, la cuarta ley determina que una secuencia de sobrescrituras es equivalente a la última escritura en la secuencia.

Observación 15. Se considera de especial relevancia el destacar aquí el trabajo de Gibbons y Hinze [39, 37], en el que introduce una aproximación para recuperar el *razonamiento ecuacional* –natural en el paradigma funcional– cuando se lidia con efectos, que hasta la fecha requerían de un tratamiento especial. Dicha aproximación se apoya directamente en la utilización de teorías algebraicas como `MonadState` para modelar los efectos y razonar sobre ellos explotando sus leyes.

Como se puede inferir, la mónada estado se posiciona como el candidato idóneo para instanciar la interfaz provista por `MonadState`. Sin embargo, se utilizará la mónada generada por el transformador `StateT` en vistas a ser más generales[19], cuya instancia se muestra a continuación:

```
implicit def stateTMonadState[S, M[_]: Monad] =
  new MonadState[StateT[M, S, ?], S] {
    def get: StateT[M, S, S] = StateT(s => point(s, s))
    def put(s: S): StateT[M, S, Unit] = StateT(_ => point(s, ()))
  }
```

Esta instancia permite lidiar con una pila donde el efecto más exterior se corresponde con el de la manipulación de estado. Sin embargo, esta instancia por sí sola no permite desplegar el acceso uniforme perseguido por MTL, ya que se desea poder utilizar esta interfaz para cualquier pila conteniendo el efecto de estado sin importar su posición en la misma. En otras palabras, la instancia anterior se podría entender como el caso base en la resolución de implícitos, pero también es necesario contar con otras instancias que contemplen los posibles pasos inductivos. Por ejemplo `OptionT` puede implementar la interfaz de

[19] Podemos recuperar `State` a partir de `StateT`, usando la mónada identidad `Id` como efecto asociado.

`MonadState`, siempre y cuando su efecto asociado contenga el efecto de estado, es decir, si éste fuese a su vez instancia de `MonadState`.

```
def optionTMonadState[S, M[_]: MonadState[?[_], S]] =
  new MonadState[OptionT[M, ?], S] {
    def get: OptionT[M, S] = liftM(MonadState[M, S].get)
    def put(s: S): OptionT[M, Unit] = liftM(MonadState[M, S].put(s))
  }
```

Esta instancia resulta trivial, simplemente se recurre a `liftM` para adaptar los programas internos, utilizando la evidencia de `MonadState` que se encarga de generarlos. Al igual que se ha hecho con el efecto de estado, se podría aplicar un proceso análogo para el efecto asociado a los errores. Se muestra una simplificación de la interfaz que ofrece MTL para esta tarea:

```
trait MonadError[M[_]] extends Monad[M] {
  def raiseError(e: String): M[Unit]
}
```

De nuevo, sería necesario ofrecer la instancia que actúa como caso base, en este caso la instancia para `OptionT` y la instancia para el paso inductivo, es decir, la instancia para `StateT` siempre y cuando contenga el efecto de error en la computación asociada. No se muestran por brevedad. Ahora sí, es posible reimplementar el método de evaluación en términos de las nuevas abstracciones:

```
def eval[M[_] : MonadState[?[_], Int] : MonadError](t: Expr): M[Int] = t match {
  case Con(a) => point(a)
  case Div(u, v) => for {
    a ← eval(u)
    b ← eval(v)
    _ ← if (b == 0) raiseError("div␣by␣0") else modify(_ + 1)
  } yield a / b
}
```

El código resultante es muy similar al que se obtuvo en la sección anterior, aunque existen ciertos matices diferenciadores que merece la pena destacar. En primer lugar, esta función no se compromete con ningún orden para la pila de efectos. La signatura simplemente indica que esta computación es capaz de acceder a un estado de tipo `Int` y que además puede producir errores. Es el programador el que se encargará de determinar dicho orden de los efectos al invocar a este método, delegando al compilador la tarea de encontrar las instancias correspondientes. El segundo elemento diferenciador es que `modify` no necesita ninguna adaptación con `liftM`. La principal ventaja que supone este acceso uniforme es que añadir nuevos efectos es tan sencillo como indicarlo en la signatura y no supondría ninguna modificación en el código, más allá de la nueva funcionalidad que se desee introducir para dicho efecto. Se recomienda al lector interesado acudir al tutorial recogido en [40], que contiene numerosos ejemplos de efectos para resolver la evaluación de términos más complejos.

Observación 16. Uno de los argumentos en la línea de *"MTL is not a monad transformer library"* reside en que las teorías algebraicas dispuestas en MTL no necesariamente tienen que estar instanciadas por transformadores de mónadas. Por ejemplo, se podría ofrecer una instancia de `MonadState` para acceder y manipular cierto estado alojado en una base de datos que cumpla con las leyes impuestas por esta abstracción.

Lamentablemente, este estilo de programación no está exento de problemas. En primer lugar, la resolución de type classes –a la que algunos se refieren como

resolución estilo *Prolog*– es compleja y puede producir errores. En particular, Haskell tiene ciertas dificultades para referirse a un efecto cuando la pila tiene varias ocurrencias del mismo, aunque el contexto sea claro al respecto[20]. Por su parte, Scala tiene ciertas dificultades para resolver instancias de type constructor classes, por lo que en muchas ocasiones es necesario construir las evidencias requeridas manualmente. En segundo lugar, y aunque aquí sólo se han mostrado un par de interfaces, la adición de nuevos efectos es un proceso verdaderamente tedioso que requiere añadir multitud de instancias: el caso base para la nueva teoría algebraica, los casos inductivos que instancian la nueva teoría algebraica para el resto de transformadores y los casos inductivos del resto de las teorías algebraicas ya existentes para el nuevo transformador.

2.4. Typed tagless final

Los lenguajes específicos de dominio (DSLs) han adquirido una gran notoriedad entre la comunidad de programadores, postulándose como una alternativa a los lenguajes de propósito general, donde destacan lenguajes tan extendidos como HTML o SQL. Existen diferentes aproximaciones para implementar DSLs, como la externa (o *standalone*) y la interna (o *embedded*). En este trabajo nos centraremos en ésta última (eDSLs), en la que un lenguaje objeto se incrusta o embebe sobre un lenguaje host. La flexibilidad y modularidad propias de los lenguajes funcionales les atribuyen unas capacidades idóneas para ejercer como lenguajes host [60]. De hecho, citando a Wadler[21]: "A functional language is a domain-specific language for creating domain-specific languages".

Esta sección introduce *typed tagless final* [13], una aproximación muy conveniente para implementar eDSLs y sus intérpretes asociados. En primer lugar se mostrará una breve comparación entre las aproximaciones inicial y final, utilizando un lenguaje de primer orden muy sencillo. Después se abordará la extensibilidad de la aproximación, donde será necesario introducir el *expression problem* como motivación. También se introducirá la normalización de expresiones utilizando transformaciones a priori no composicionales. Finalmente se introducirá un lenguaje de orden superior para mostrar el potencial de la aproximación al completo. En esencia, se seguirá un estilo muy similar a [70], donde se destacarán los aspectos más relevantes en el contexto de esta tesis y donde se describirán las peculiaridades propias de la adaptación de estas técnicas al lenguaje de programación Scala, que utilizaremos en esta sección para guiar las explicaciones.

2.4.1. Aproximaciones inicial y final

La técnica más extendida para embeber un lenguaje en otro es mediante tipos algebraicos de datos, de lo que ya se vio un ejemplo con el ADT `Expr` que se presentó al principio de la sección 2.3. Este lenguaje se vuelve a mostrar a

[20] Por ejemplo, si la pila manipula un estado de tipo `Int` y otro, más anidado, de tipo `Boolean`, se producirá un error cuando tratemos de invocar un `get` que recupera un `Boolean`.

[21] En particular, esta frase aparece en las diapositivas de una charla asociada al trabajo de investigación realizado en [93] (`https://homepages.inf.ed.ac.uk/wadler/papers/qdsl/googlex.pdf`)

continuación, aunque la división (`Div`) será reemplazada por una suma (`Add`)[22], ya que esta sección no pretende centrarse en la gestión de errores:

```
sealed abstract class Expr
case class Lit(i: Int) extends Expr
case class Add(x: Expr, y: Expr) extends Expr
```

Una expresión sencilla como $(1 + ((2 + 3) + 4))$ se representaría mediante la expresión:

```
val ei1: Expr =
  Add(Lit(1), Add(Add(Lit(2), Lit(3)), Lit(4)))
```

La evaluación directa de esta expresión resulta ser una tarea trivial, donde no se requiere gestionar errores o contar el número de sumas, como sí ocurría en los evaluadores mostrados en la sección 2.3:

```
def eval(e: Expr): Int = e match {
  case Lit(i) => i
  case Add(x, y) => eval(x) + eval(y)
}
```

La evaluación de `ei1` con la nueva función se recoge a continuación:

```
val res: Int = eval(ei1)
// res: Int = 10
```

El comentario en la última línea indica el valor contenido en `res`, es decir, el resultado de la evaluación.

En lugar de utilizar ADTs podríamos representar este lenguaje utilizando funciones que trabajan directamente sobre el dominio semántico de la evaluación:

```
type Repr = Int
def lit(i: Int): Repr = i
def add(x: Repr, y: Repr): Repr = x + y
```

Como se puede apreciar se introduce un type alias `Repr` que determina el objetivo de la evaluación y funciones que trabajan en torno a él. Con estas funciones disponibles, podríamos representar la expresión análoga a `ei1` de la siguiente manera:

```
val ef1: Repr =
  add(lit(1), add(add(lit(2), lit(3)), lit(4)))
```

La expresión resultante es prácticamente idéntica a la anterior, aunque es destacable la nueva capitalización para referirnos a los términos. Si tenemos en cuenta que la representación se corresponde con la semántica, no se requiere ninguna función de evaluación, aunque se introducirá el análogo para facilitar la comparación:

```
def eval(r: Repr): Int = r
```

De hecho, la definición de `eval` en este escenario se corresponde con la función identidad. Para recuperar el resultado de la evaluación procedemos de la misma manera que en el caso inicial:

```
val res: Int = eval(ef1)
// res: Int = 10
```

[22] De hecho, el lenguaje resultante tras este cambio suele conocerse informalmente como *Hutton's Razor*, cuya primera aparición se recoge en [62].

Aquí se puede apreciar cómo ambas aproximaciones se evaluan al mismo resultado.

La segunda aproximación se denomina informalmente como *final*, como una contraposición al término *inicial*. Aunque el término inicial sí que se inspira en la Teoría de Categorías, el término final en este contexto es totalmente independiente de su definición categórica [65]. De hecho, tal y como muestra Kiselyov en [70] las aproximaciones inicial y final resultan ser isomórficas y por tanto ambas resultan ser iniciales desde una perspectiva matemática. En las próximas secciones también se contemplaran los aspectos *typed* y *tagless* para motivar el nombre completo de la aproximación.

2.4.2. Extensibilidad y el problema de la expresión

El *problema de la expresión* (EP), formulado por Wadler en 1998, tiene por objetivo la extensión de los casos que conforman un tipo de datos y la extensión de las funciones que trabajan sobre él, sin la posibilidad de recompilar el código existente. Afortunadamente, la combinación del paradigma funcional y el paradigma de orientación a objetos introducido por Scala nos ofrece una gran riqueza para ilustrar este problema. El objetivo final de esta sección es el de mostrar que el estilo tagless-final es capaz de solventarlo y por tanto tiene atribuido una gran capacidad de extensibilidad.

En primer lugar, proporcionaremos una interpretación adicional para `Expr`, que nos permitirá traducir expresiones en una cadena de texto con una notación más matemática, como la que se mostró anteriormente $(1 + ((2 + 3) + 4))$. Este tipo de intérprete suele ser conocido como *pretty printer*, del que mostramos su implementación justo a continuación:

```
def pretty(e: Expr): String = e match {
  case Lit(i) => s"$i"
  case Add(x, y) => s"(${pretty(x)} + ${pretty(y)})"
}
```

Como se puede apreciar, esta definición no requiere la recompilación del código existente, ya que no efectúa ningún cambio sobre él. El siguiente paso es el de añadir un nuevo caso en el tipo de datos, es decir, añadir una nueva primitiva para el lenguaje `Expr`. Por ejemplo, podría resultar de interés añadir un nuevo término que permita la multiplicación de expresiones, para lo que sería necesario extender `Expr` de la siguiente manera:

```
sealed abstract class Expr
case class Lit(i: Int) extends Expr
case class Add(x: Expr, y: Expr) extends Expr
case class Mul(x: Expr, y: Expr) extends Expr
```

La adición de un nuevo caso `Mul` deriva en la recompilación del módulo en el que se encuentra `Expr`, con lo que surge el EP. Sin embargo, éste es el menor de los problemas. Añadir un nuevo caso rompe el código de los intérpretes existentes, o al menos rompe su totalidad, por no contemplar el nuevo caso en el pattern matching. La adaptación de `eval` incluye un caso para el nuevo término:

```
def eval(e: Expr): Int = e match {
  case Lit(i) => i
  case Add(x, y) => eval(x) + eval(y)
  case Mul(x, y) => eval(x) * eval(y) // nuevo caso
}
```

Lo mismo ocurre para la extensión de `pretty`:

```
def pretty(e: Expr): String = e match {
  case Lit(i) => s"$i"
  case Add(x, y) => s"(${pretty(x)} + ${pretty(y)})"
  case Mul(x, y) => s"(${pretty(x)} * ${pretty(y)})" // nuevo caso
}
```

La nueva instrucción nos permite componer expresiones ligeramente más complejas, donde también se contempla la multiplicación:

```
val ei2: Expr = Mul(Lit(2), ei1)
val res1: Int = eval(ei2)
// res1: Int = 20
val res2: String = pretty(ei2)
// res2: String = (2 * (1 + ((2 + 3) + 4)))
```

Este ejemplo permite corroborar la idea de que la programación funcional permite extender fácilmente nuevas funciones en torno a un ADT, pero sufre notablemente cuando se añaden nuevos casos sobre el tipo de datos.

En vez de saltar directamente a la aproximación final, y aprovechando la selección de Scala como lenguaje para guiar las explicaciones, se expone el EP desde la perspectiva de la orientación a objetos [102]. Para ello, se proporciona la figura 2.18 en el que se muestra una clase abstracta `Expr` de la que heredan los casos `Lit` y `Add`, que instancian la evaluación de cada tipo de objeto para el método `eval`. De nuevo se abordará la extensión de `Expr` con un término para multiplicar y con un pretty printer.

```
abstract class Expr {
  def eval: Int
}

case class Lit(i: Int) extends Expr {
  def eval: Int = i
}

case class Add(x: Expr, y: Expr) extends Expr {
  def eval: Int = x.eval + y.eval
}
```

Figura 2.18: Diseño de expresión en orientación a objetos

En este diseño añadir un nuevo caso para nuestro lenguaje es trivial. Simplemente se requiere incluir una nueva clase que herede de `Expr` y extienda su interfaz:

```
case class Mul(x: Expr, y: Expr) extends Expr {
  def eval: String = x.eval * y.eval
}
```

Este resultado muestra que la extensión de nuevos casos en un diseño orientado a objetos es trivial y no requiere la recompilación de los módulos existentes.

Sin embargo, este tipo de diseños sufren a la hora de incluir nuevas interpretaciones, donde se manifiesta el EP. La figura 2.19 recoge la adaptación del pretty printer. Tras añadir el nuevo intérprete en la interfaz de `Expr` todo el código existente se rompe, requiriendo que cada caso incluya una implementación para el nuevo componente abstracto `pretty`. Por lo tanto, la facilidad de

```
abstract class Expr {
  def eval: Int
  def pretty: String
}

case class Lit(i: Int) extends Expr {
  def eval: Int = i
  def pretty: String = s"$i"
}

case class Add(x: Expr, y: Expr) extends Expr {
  def eval: Int = x.eval + y.eval
  def pretty: String = s"(${x.pretty} + ${y.pretty})"
}

case class Mul(x: Expr, y: Expr) extends Expr {
  def eval: Int = x.eval * y.eval
  def pretty: String = s"(${x.pretty} * ${y.pretty})"
}
```

Figura 2.19: Extensión de diseño OO con `pretty`

extensibilidad es justo la contraria a la que se producía en un diseño funcional convencional.

Se repite ahora el mismo experimento para la aproximación final que se presentó en la sección 2.4.1. En primer lugar, se procederá con la extensión del término que recoge las multiplicaciones. Añadir esta nueva entrada simplemente consiste en la inclusión de una nueva función `mul`:

```
def mul(x: Repr, y: Repr): Repr = x * y
```

La inclusión de la nueva función no requiere modificar, ni tan siquiera recompilar el código existente. Con esta nueva función es posible componer una expresión análoga a `ei2`:

```
val ef2: Repr = mul(lit(2), add(lit(1), add(add(lit(2), lit(3)), lit(4))))
val res: Int = eval(ef2)
// res: Int = 20
```

Tal y como se puede apreciar, el resultado de la expresión es consistente.

El siguiente paso sería el de introducir una nueva interpretación que se correspondiese con el pretty printer. Lamentablemente, la extensión de un nuevo intérprete es difícil de vislumbrar dada la situación actual. Todo apunta a que se debería configurar una representación `Repr` que apuntase a un `String`, pero esto rompería las funciones existentes que trabajan sobre el dominio semántico de la evaluación. La solución pasa por parametrizar el tipo de representación, mediante una type class (ver apéndice A.1). Las próximas líneas adaptan el código existente al nuevo embedding, comenzando por la introducción de la type class:

```
trait HuttonSym[Repr] {
  def lit(i: Int): Repr
  def add(x: Repr, y: Repr): Repr
}
```

Las definiciones que componen la interfaz son exactamente idénticas a las que se presentaron anteriormente, pero la representación deja de ser un type alias

y se convierte en el parámetro de la clase. Merece la pena destacar que *Sym* viene de *Symantics*, un término que pretende fusionar *Syntax* y *Semantics*. De hecho, la clase en sí misma refleja la sintaxis y el sistema de tipos, mientras que sus instancias determinan la semántica. Por ejemplo, la semántica de evaluación viene determinada con esta instancia:

```
implicit object R extends HuttonSym[Int] {
  def lit(i: Int): Int = i
  def add(x: Int, y: Int): Int = x + y
}
```

Este intérprete suele denominarse meta-circular, ya que convierte los términos del lenguaje objeto en términos del lenguaje host: los literales del lenguaje Hutton se traducen a valores enteros de Scala y la suma de expresiones se implementa en términos de la suma de Scala. El nombre *R* suele ser habitual para referirse a este tipo de instancias meta-circulares. La adaptación de `ei1` pasa a convertirse en un método parametrizado por el tipo de representación, que debe ser instancia de `HuttonSym`, cuya adaptación se refleja en este fragmento de código:

```
def ef1[Repr](implicit H: HuttonSym[Repr]): Repr =
  H.add(H.lit(1), H.add(H.add(H.lit(2), H.lit(3)), H.lit(4)))
```

Esta definición es notablemente más compleja que la que se mostró con anterioridad, principalmente debido a la nueva signatura, que requiere evidencias implícitas, y al ruidoso acceso a los términos definidos en la evidencia *H*. Por suerte, Scala proporciona mecanismos para introducir el azúcar sintáctico que permite mitigar esta situación. Para este caso concreto, resultaría ser de utilidad introducir los siguientes conversores:

```
implicit def lit[Repr](i: Int)(implicit ev: HuttonSym[Repr]): Repr =
  ev.lit(i)

implicit def add[Repr](x: Repr, y: Repr)(implicit ev: HuttonSym[Repr]): Repr =
  ev.add(x, y)
```

Con ellos, se podría reimplementar `ef1` de una manera más natural, tal y como se muestra en esta nueva versión, que utiliza la notación de *context bound* para añadir la dependencia:

```
def ef1[Repr: HuttonSym]: Repr =
  add(lit(1), add(add(lit(2), lit(3)), lit(4)))
```

Para llevar a cabo la evaluación de la expresión, simplemente se requiere indicar el tipo de representación a utilizar y Scala se encargará de proporcionar la interpretación correspondiente de manera implícita:

```
val res = ef1[Int]
// res: Int = 10
```

Con esto, se recupera el código existente adaptado al nuevo embedding basado en una type class. Los siguientes párrafos abordan la extensión de términos e intérpretes para la nueva configuración.

Se comenzará con la inclusión del nuevo intérprete. Ya se ha visto que la semántica asociada a los términos de un lenguaje se determina mediante la instancia de la type class que los recoge. Por tanto la única tarea a llevar a cabo para abordar esta extensión es la de añadir una nueva instancia que contemple el pretty printer:

```
implicit object Pretty extends HuttonSym[String] {
  def lit(i: Int): String = s"$i"
  def add(x: String, y: String): String = s"($x␣+␣$y)"
}
```

Obviamente, la introducción de este intérprete no rompe el código existente ni requiere de su recompilación. Proporcionando la nueva representación a la expresión `ef1` se consigue su traducción a texto:

```
val res = ef1[String]
// res: String = "(1 + ((2 + 3) + 4))"
```

Ahora se procederá con la inclusión del nuevo término que dote al lenguaje con capacidades de multiplicación. En vez de añadir una nueva entrada para este término en `HuttonSym`, lo que forzaría la recompilación de este componente, se propone incluir el nuevo término en una clase independiente:

```
trait MulSym[Repr] {
  def mul(x: Repr, y: Repr): Repr
}
```

Esta clase únicamente recoge el término `mul` como parte de su interfaz abstracta. Como puede resultar evidente, las instancias de la type class también serán independientes de las instancias de `HuttonSym`, es decir, no se requiere ninguna modificación sobre los intérpretes existentes. A continuación se muestra la instancia meta-circular para el lenguaje de multiplicación:

```
implicit object R extends MulSym[Int] {
  def mul(x: Int, y: Int): Int = x * y
}
```

y la instancia asociada al pretty printer:

```
implicit object Pretty extends MulSym[String] {
  def mul(x: String, y: String): String = s"($x␣*␣$y)"
}
```

Adelantando la composición de expresiones en una notación más compacta y sencilla, se proporciona el azúcar sintáctico asociado al nuevo término:

```
implicit def mul[Repr](x: Repr, y: Repr)(implicit ev: MulSym[Repr]): Repr =
  ev.mul(x, y)
```

La definición análoga a `ef2` depende de los términos definidos en ambos bloques de symantics, aspecto que se manifiesta en la signatura de la siguiente manera:

```
def ef2[Repr : HuttonSym : MulSym]: Repr =
  mul(lit(2), ef1)
```

La sintaxis del context bound nos indica que `Repr` debe ser una instancia tanto de `HuttonSym` como de `MulSym`. Merece la pena destacar que el compilador de Scala pasa la evidencia de `HuttonSym` a `ef1` de forma implícita. Gracias a los nuevos intérpretes y reutilizando los ya existentes, es posible traducir la expresión genérica `ef2` a su evaluación y a su representación textual:

```
val res1 = ef2[Int]
// res1: Int = 20

val res2 = ef2[String]
// res2: String = (2 * (1 + ((2 + 3) + 4)))
```

Por lo tanto, el estilo final utilizando type classes nos permite extender tanto los términos del lenguaje como sus interpretaciones, sin la necesidad de modificar o recompilar el código existente, ofreciendo una solución muy elegante al EP.

Observación 17. En esta sección se ha podido apreciar que las type classes son un elemento central para tagless-final, al igual que sucedía en MTL. Sin embargo, las teorías algebraicas de MTL clasifican tipos de computaciones, mientras que el parámetro `Repr` de la clase `HuttonSym` se ha instanciado con tipos básicos como `Int` o `String`, que evidentemente, no tienen ninguna computación asociada. En la siguiente sección se seguirá extendiendo esta discusión.

2.4.3. Lenguajes tipados e intérpretes *tagged*

Hasta ahora se ha estado utilizando un lenguaje objeto muy sencillo, donde todas las expresiones denotan el mismo tipo. Esta sección pretende demostrar los problemas que surgen cuando se contemplan tipos adicionales en el lenguaje objeto. Finalmente se presentará la solución completa propuesta por la aproximación typed tagless final, que permite la implementación del sistema de tipos.

El lenguaje que se propone para guiar las explicaciones en esta sección se recoge en el siguiente módulo de symantics:

```
trait BaseSym[Repr] {
  def int(i: Int): Repr
  def bool(b: Boolean): Repr
  def greaterThan(x: Repr, y: Repr): Repr
  def ifs(c: Repr, _then: Repr, _else: Repr): Repr
}
```

El lenguaje no es muy diferente al anterior, la principal novedad es la introducción de literales para varios tipos base, como enteros (`int`) y booleanos (`bool`). También se muestran un par de combinadores que permiten jugar con ellos, como la comparación de enteros (`greaterThan`), que recibe un par de enteros y devuelve un booleano, o la estructura de control *if-then-else* (`ifs`), que recibe una condición booleana y la expresión que se debe ejecutar en caso de que la condición se cumpla o no se cumpla.

Con este lenguaje es posible escribir expresiones como la que se presenta a continuación, donde se asume la existencia de azúcar sintáctico análogo al de los lenguajes anteriores, que no se muestra por brevedad:

```
def ef3[Repr : BaseSym]: Repr =
  ifs(greaterThan(int(4), int(2)), _then = int(1), _else = int(0))
```

El intérprete meta-circular asociado a esta expresión debería de producir la evaluación de la expresión Scala: `if (4 > 2) 1 else 0`. Sin embargo, la instancia de `BaseSym` que permita su generación no resulta trivial. ¿Cuál sería la representación que nos permitiría evaluar tanto enteros como booleanos? Parece que la alternativa más viable pasa por introducir un tipo donde se contemplan ambas variantes:

```
sealed abstract class Tag
case class IntTag(i: Int) extends Tag
case class BoolTag(b: Boolean) extends Tag
```

Esto es esencialmente un ADT que proporciona un caso (o *tag*) para producir la evaluación de un entero (`IntTag`) y otro caso para producir la evaluación de un booleano (`BoolTag`). Con el nuevo tipo de datos se dispone de un marco para

poder llevar a cabo una instanciación del lenguaje, cuya implementación aparece en la figura 2.20. Como se puede apreciar, es necesario realizar pattern matching sobre la evaluación para recoger el tipo de valor esperado en cada situación.

```
implicit object Tagged extends BaseSym[Tag] {

  def int(i: Int): Tag = IntTag(i)

  def bool(b: Boolean): Tag = BoolTag(b)

  def greaterThan(x: Tag, y: Tag): Tag = (x, y) match {
    case (IntTag(i), IntTag(j)) => BoolTag(i > j)
  }

  def ifs(c: Tag, _then: Tag, _else: Tag): Tag = c match {
    case BoolTag(b) => if (b) _then else _else
  }
}
```

Figura 2.20: Intérprete *tagged* para la evaluación de expresiones.

Con la nueva semántica es posible evaluar el resultado de `ef3`, siguiendo el patrón habitual, en este caso particular determinando `Tag` como representación:

```
val res = ef3[Tag]
// res: Tag = IntTag(0)
```

El resultado nos muestra que la evaluación de la expresión es de tipo entero y que su valor es `0`. Aunque la solución es correcta, esta aproximación tiene una limitación muy grande: el embedding no es ajustado (*tight*) ya que permite muchas más expresiones que las que incluye el lenguaje. Por ejemplo, esta expresión, donde se utiliza un entero en lugar de un booleano como condición, compila correctamente:

```
def ef4[Repr : BaseSym]: Repr =
  ifs(int(2), _then = int(0), _else = int(1))
```

De hecho, este problema también se manifiesta en el intérprete, donde sólo se contemplan los casos esperados. Esto deriva en pattern matching no exhaustivo, es decir, introduce parcialidad. Adicionalmente a estos problemas, la introducción de tipos unión para dar soporte a la evaluación de lenguajes más complejos suele derivar en una acusada caída del rendimiento en la interpretación, consecuencia de la continua construcción/destrucción de objetos.

La solución que ofrece el estilo typed tagless final para esta situación pasa por parametrizar la representación con el tipo que denota. Como resultado, en lugar de una type class, el fichero de symantics se define en términos de una type constructor class:

```
trait BaseSym[Repr[_]] {
  def int(i: Int): Repr[Int]
  def bool(b: Boolean): Repr[Boolean]
  def greaterThan(x: Repr[Int], y: Repr[Int]): Repr[Boolean]
  def ifs[A](c: Repr[Boolean], _then: Repr[A], _else: Repr[A]): Repr[A]
}
```

La nueva versión de `BaseSym` no sólo contempla la sintaxis del lenguaje, sino que también implementa su sistema de tipos. Por ejemplo, `greaterThan` manifiesta

que las expresiones de entrada deben denotar un tipo entero, mientras que la salida denotará un booleano. La definición de las expresiones genéricas debe adaptarse a la nueva situación, donde se hace evidente que la representación es un constructor de tipos y donde se refleja el valor denotado por el resultado:

```
def ef3[Repr[_]: BaseSym]: Repr[Int] =
  ifs(greaterThan(int(4), int(2)), _then = int(1), _else = int(0))
```

Afortunadamente, el cuerpo de la definición se mantiene exactamente igual[23].

```
implicit object Tagless extends BaseSym[λ[x => x]] {

  def int(i: Int): Int = i

  def bool(b: Boolean): Boolean = b

  def greaterThan(x: Int, y: Int): Boolean = x > y

  def ifs[A](c: Boolean, _then: A, _else: A): A =
    if (c) _then else _else
}
```

Figura 2.21: Intérprete meta-circular *tagless*.

La instancia que define el intérprete meta-circular se muestra en la figura 2.21. La representación que se utiliza en este tipo de intérpretes suele ser `Id`, es decir, cada término devuelve un valor del tipo que denota. En vez de `Id`, se utilizará una función de tipos *lambda* que representa la misma idea. El intérprete resultante es realmente simple, sin etiquetas y total. Se puede utilizar para efectuar la evaluación de `ef3`:

```
val res = ef3[λ[x => x]]
// res: Int = 0
```

Merece la pena destacar que el resultado no está envuelto en ninguna etiqueta, sino que se trata de un entero plano. La introducción del sistema de tipos en los términos deriva en un embedding tight, que sólo permite componer expresiones válidas, con lo que la adaptación de `ef4` produciría un error en el compilador:

```
def ef4[Repr[_]: BaseSym]: Repr[Int] =
  ifs(int(2), _then = int(0), _else = int(1))
```

En este caso particular, el compilador de Scala marca `int(2)` e indica que se esperaba una expresión de tipo `Repr[Boolean]` pero que se ha encontrado una expresión de tipo `Repr[Int]` en su lugar.

Es importante destacar que el nuevo cambio preserva las garantías de extensibilidad que se presentaron en la sección 2.4.2. A modo de ejemplo, introducimos el intérprete pretty printer en la figura 2.22. Es interesante analizar la representación utilizada, que se corresponde con una función de tipos constante, que siempre devuelve un String, con lo que la instancia ignora por completo los tipos definidos en el módulo de symantics. Desde esta perspectiva y para este caso concreto, los tipos del lenguaje objeto se podrían entender como meros *phantom types* [32]. El siguiente fragmento de código muestra la traducción a texto de la expresión `ef3`:

[23] Las definiciones implícitas que introducen el azúcar sintáctico también deben adaptarse a trabajar con un constructor de tipos.

```
implicit object Pretty extends BaseSym[Lambda[x => String]] {

  def int(i: Int): String = s"$i"

  def bool(b: Boolean): String = s"$b"

  def greaterThan(x: String, y: String): String = s"$x > $y"

  def ifs[A](c: String, _then: String, _else: String): String =
    s"if ($c) ${_then} else ${_else}"
}
```

Figura 2.22: Pretty printer para lenguaje tipado.

```
val res: String = ef3[λ[x => String]]
// res: String = if (4 > 2) 1 else 0
```

Aquí se aprecia que el pretty printer nos muestra la expresión traducida a una notación más legible, que se corresponde con una expresión *if-then-else* de Scala.

Observación 18. A pesar de haber introducido un constructor de tipos `Repr[_]` como parámetro de la type class `BaseSym`, es importante destacar que en este caso tampoco se están modelizando tipos de computaciones, sino tipos de representaciones. Esta noción de representación resulta ser más general y ofrece un marco para la aplicación de optimizaciones agresivas.

2.4.4. Lenguajes de orden superior

Esta sección muestra el embedding de un lenguaje más ambicioso, donde se recojan tipos más complejos, para mostrar el potencial de la aproximación typed tagless final. Para la tarea se utilizará *higher-order abstract syntax* (HOAS) como lenguaje objeto, en el que las funciones se tratan como *ciudadanos de primera clase*. El clásico *Why Functional Programming Matters?* de Hughes [61] ya ilustraba este aspecto como uno de los mecanismos de modularidad más relevantes de este paradigma. No obstante, al final de la sección, se discutirá por qué no siempre resulta una buena práctica introducir el *cálculo lambda* para embeber un lenguaje que requiera de funciones.

La sintaxis y el sistema de tipos de HOAS se introducen mediante el siguiente módulo de symantics:

```
trait HoasSym[Repr[_]] extends HuttonSym[Repr] {
  def lam[A, B](f: Repr[A] => Repr[B]): Repr[A => B]
  def app[A, B](f: Repr[A => B], a: Repr[A]): Repr[B]
}
```

Esencialmente, `HoasSym` extiende[24] el lenguaje Hutton's Razor con dos nuevos términos[25]. El primero de ellos (`lam`) introduce una expresión lambda, donde se introduce el tipo función como dominio semántico abstracto; el segundo término

[24] No es imprescindible que `HoasSym` extienda `HuttonSym`. De hecho, se podría definir como un módulo independiente y que fuese la propia lógica de negocio la que se encargase de componer los módulos requeridos. No obstante, se adopta este estilo por ser más homogéneos con el resto de las secciones.

[25] Aquí se asumirá una versión de `HuttonSym` que ha sido parametrizada mediante un constructor de tipos `Repr[_]`, en línea con `BaseSym` de la sección anterior.

(app) se corresponde con la aplicación de funciones. Este lenguaje permite la composición de expresiones que denoten funciones, como por ejemplo:

```
def ef5[Repr[_]: HoasSym]: Repr[Int => Int] =
  lam(x => add(x, int(2)))
```

Esta definición se correspondería con la expresión `x => x + 2` en Scala, la cual se podría generar mediante el intérprete meta-circular (figura 2.23). La implementación es trivial: `lam` se convierte en la función identidad y `app` simplemente aplica la función y el argumento recibidos como parámetros. Para llevar a cabo la evaluación de `ef5`, simplemente se concreta la representación del intérprete meta-circular:

```
val res = ef5[λ[x => x]].apply(2)
// res: Int = 4
```

Como las funciones en Scala no se pueden imprimir, se muestra la aplicación del argumento `2` sobre el resultado, que produce `4` (`2 + 2`).

```
implicit object R extends HoasSym[λ[x => x]] {

  def int(i: Int): Int = i

  def add(x: Int, y: Int): Int = x + y

  def lam[A, B](f: A => B): A => B = f

  def app[A, B](f: A => B, a: A): B = f(a)
}
```

Figura 2.23: Intérprete meta-circular para HOAS

Más interesante resulta el caso del pretty printer, que se muestra en la figura 2.24, cuya implementación no resulta ser tan trivial. El primer aspecto que debería llamar la atención es la representación `λ[x => Int => String]`. Con ella se pretende que las computaciones tengan acceso a un contador, que es necesario para poder proporcionar nombres *fresh* para los parámetros introducidos por las expresiones lambda. La implementación del literal `int` es sencilla, simplemente se ignora el contador. Para los casos asociados a `add` y `app` es necesario pasar el contador actual a los diferentes componentes de la expresión. Finalmente, `lam` es el encargado de proporcionar un nombre fresh para el parámetro de la expresión lambda y de suplementar la nueva versión del contador a la expresión que contiene el cuerpo de la función. La traducción de `ef5` a su versión textual se muestra aquí:

```
val res = ef5[λ[x => Int => String]].apply(0)
// res: String = x0 => (x0 + 2)
```

Como se puede observar, se proporciona el argumento `0` al resultado de la traducción, que se corresponde con el valor inicial para el contador.

Aunque la potencia del lambda calculus es bien reconocida por todo programador funcional, la interpretación de `lam` y `app` puede volverse muy compleja en determinadas infraestructuras. Este aspecto ya se ha podido apreciar ligeramente en la interpretación del pretty printer (figura 2.24), donde la implementación de `lam` no resulta ser tan directa como la del resto de términos. Este aspecto se

```
implicit object Pretty extends HoasSym[λ[x => Int => String]] {

  def int(i: Int): Int => String = const(s"$i")

  def add(x: Int => String, y: Int => String): Int => String = { cnt =>
    s"(${x(cnt)} + ${y(cnt)})"
  }

  def lam[A, B](f: (Int => String) => (Int => String)): Int => String = { cnt =>
    val x = s"x$cnt"
    s"$x => ${f(const(x))(cnt + 1)}"
  }

  def app[A, B](f: Int => String, a: Int => String): Int => String = { cnt =>
    s"${f(cnt)}(${a(cnt)})"
  }
}
```

Figura 2.24: Intérprete pretty printer para HOAS

tiene muy en cuenta en el diseño de SQUR [73], donde se pone mucho énfasis en mantener un lenguaje de primer orden[26]. El autor argumenta que es recomendable prescindir del HOAS en aquellas situaciones en las que el lenguaje destino no contenga expresiones lambda, como bien podría ser el caso de SQL. A pesar de esto, siempre sería posible utilizar la potencia del lenguaje host para enriquecer las definiciones. Por ejemplo, la expresión `ef5` se podría haber implementado mediante una simple función de Scala:

```
def ef5[Repr[_]: HoasSym]: Repr[Int] => Repr[Int] =
  x => add(x, int(2))
```

Aunque esto podría introducir algunas limitaciones (por ejemplo, ¿cómo se (de)serializa esta definición?), mantener un lenguaje de primer orden podría resultar conveniente para un gran número de situaciones.

2.5. Language-Integrated Query

Esta sección pretende ilustrar los fundamentos de las técnicas de LINQ basadas en comprehensions mediante un ejemplo que contempla un conjunto de parejas donde cada miembro que la forma tiene asociado un nombre y una edad:

```
class Couple(her: String, him: String)
class Person(name: String, age: Int)
```

Esta forma de representar la información es *plana* o *relacional*: las entidades no contienen referencias anidadas a otras entidades, sino que utilizan claves foráneas para establecer relaciones entre ellas. En particular, `her` y `him` deberían contener el mismo valor que el `name` de alguna persona, campo que representa un identificador único.

Dada una colección de parejas y una colección de personas, debería de ser posible obtener los nombres de aquellas mujeres que forman parte de una pareja

[26] Aunque el artículo en cuestión no incide demasiado en este aspecto, Kiselyov pone bastante empeño en clarificar la ausencia de lambda calculus en SQUR en esta charla realizada en el EPFL (`https://slideshot.epfl.ch/play/icc_oleg`). Se sigue una línea muy similar en [71], donde se evitan los combinadores monádicos para la introducción de efectos.

y que tienen menos de 50 años de edad, mediante una consulta basada en (list) comprehensions [126, 10]:

```
def under50(couples: List[Couple], people:  List[Person]): List[String] =
  for {
    c ← couples
    w ← people
    if c.her == w.name && w.age < 50
  } yield w.name
```

Ahora, usando Quill [9] se podría expresar la misma consulta de manera general. En este contexto, se precisa una *consulta genérica* como una consulta que puede ser ejecutada de manera eficiente contra almacenes de datos de diversos tipos: estructuras de datos en memoria, bases de datos relacionales, almacenes no relacionales como documentos XML/JSON, etc. El análogo genérico a `under50` se muestra a continuación:

```
val under50_quill = quote {
 for {
    c ← query[Couple]
    w ← query[Person]
    if c.her == w.name && w.age < 50
  } yield w.name
}
```

Como se puede apreciar la implementación es prácticamente idéntica a la de la versión anterior, donde la única diferencia radica en el uso de `query` para producir las expresiones que denotan listas y la introducción de un bloque `quote` que envuelve a la comprehension. De hecho, esta característica refleja una aproximación de embedding basada en QDSLs [19, 93].

Siendo genérica, esta consulta podría ser compilada a diferentes tecnologías, teniendo en cuenta las tecnologías subyacentes soportadas por el framework Quill (actualmente se soporta SQL y CQL de Cassandra). Por ejemplo, la siguiente consulta SQL es la que se generaría a partir de ella:

```
SELECT w.name
FROM Couple AS c INNER JOIN Person AS w ON c.her = w.name
WHERE w.age < 50
```

Gracias al trabajo de Cooper [23] la normalización de las consultas es un proceso que ofrece unas garantías mínimas de calidad. Esto permite que consultas a priori no eficientes —ya sea por la inexperiencia del programador o simplemente por mejorar su legibilidad— sean reducidas a su mínima expresión.

Se ha ilustrado el uso de comprehensions con un ejemplo sencillo de una consulta *flat-flat*, es decir, una consulta que recibe y devuelve tipos planos: o bien tipos base (enteros, booleanos, etc.) o bien registros que no tienen entidades anidadas. Por ejemplo, no se puede escribir una consulta SQL que devuelva una columna multivaluada. Esto muestra una importante diferencia entre SQL y un lenguaje de programación convencional, donde las estructuras anidadas están muy extendidas. Si se tiene en cuenta que el lenguaje de comprehensions, fundamentado en NRC, es capaz de lidiar con estructuras anidadas, uno podría pensar que esto podría derivar en una notable pérdida de expresividad para el caso de LINQ. Sin embargo, la literatura muestra que el uso de estructuras anidadas como valores intermedios [130], incluso con la presencia de consultas parametrizadas [23, 19], permite la generación de consultas normalizadas carentes de datos anidadas. No obstante, también sería posible contemplar consultas *flat-nested*, que se transformarían en una serie de consultas flat-flat mediante

técnicas de *query shredding* [20]. En conclusión, las comprehensions son verdaderamente convenientes desde la perspectiva de LINQ: están integradas en un amplio rango de lenguajes y suficientemente cercanas a las bases de datos relacionales como para producir consultas elegantes y eficientes.

A pesar de esto, aparecen algunos inconvenientes en la aplicación de comprehensions para LINQ que hacen que no sean idóneas para todas las situaciones, que se muestran en los siguientes puntos:

- Las comprehensions sólo permiten expresar consultas de lectura, pero las actualizaciones son igualmente importantes. Este aspecto se reconoce como un problema abierto en el campo de LINQ [16].

- El uso de estructuras anidadas intermedias y abstracción funcional en lenguajes basados en comprehensions somo Links/T-LINQ ayudan en la consecución de consultas más composicionales [19]. Sin embargo, esto es posible a expensas de una técnica de reescritura compleja, especialmente en el caso de QDSLs como T-LINQ. Algunas aproximaciones alternativas basadas en normalización por evaluación [73] alivian este problema, pero el soporte para consultas composicionales sigue siendo limitado. Esto se debe a que las comprehensions están más cerca de la notación *point-wise* propia del calculo relacional que del algebra relacional pura y tienen que lidiar con renombrado de variables, *freshness* y *scope*. Funcionalmente, ambos formalismos son igualmente expresivos, pero los combinadores *point-free* del algebra relacional son más flexibles [38]. Esta flexibilidad deriva en consultas más modulares, lo que impacta directamente en aspectos no funcionales como la reutilización y la tolerancia a cambios [104, 61].

- Existen infraestructuras de consultas que son esencialmente jerárquicas en vez de relacionales, como podría ser una base de datos NoSQL orientada a documentos, que se construye sobre fuentes de datos anidadas en JSON, XML o YAML, entre otros. La traducción de las consultas para estas infraestructuras se podría beneficiar de un modelo de consultas más algebraico y jerárquico por naturaleza.

Parte II

Ópticas y Teorías Algebraicas

Esta segunda parte de la tesis da respuesta a los objetivos 1 y 2 propuestos en la sección 1.4. Esencialmente, el capítulo 3 muestra los resultados obtenidos en [86], extendiendo algunos aspectos que no se mostraron en el artículo original por brevedad. Por ejemplo, se ofrece mayor detalle sobre la formalización de las pruebas en Coq, que a pesar de su enorme relevancia, quedaron relegadas a un segundo plano. Por otro lado, el capítulo 4 ofrece una visión detallada sobre la implementación de estas ideas en Stateless*. Esta librería escrita en Scala incluye muchas otras abstracciones experimentales y patrones recurrentes que resultaron ser de gran utilidad para el diseño de las aplicaciones que se utilizaron para validarla.*

El capítulo 5 discute cómo la aproximación propuesta aborda los problemas asociados a los repositorios funcionales (descritos en la sección 1.1), pero deriva en ciertas limitaciones que hacen que el enfoque no sea explotable desde una perspectiva industrial. Afortunadamente, la parte III, que a priori se postulaba como independiente a la aquí presente, resulta tener una importante conexión con ésta ya que ofrece una solución para dichas limitaciones. En la parte IV se ofrecerá una comparativa entre ambas aproximaciones, donde se pondrá énfasis en este aspecto.

Capítulo 3

Hacia las optic algebras: el caso de lens

Este capítulo desarrolla el **Obj. 1** (sección 1.4), con el que se pretenden establecer los pilares de la línea de investigación que busca analogías entre MTL y las ópticas, con el fin de trasladar los beneficios de estas últimas a un plano genérico. En particular, estas son las contribuciones que se recogen en este capítulo:

- Se demuestra formalmente que `MonadState` es una generalización de una lens. En particular, se presenta una nueva representación para esta abstracción, basada en la instancia de dicha teoría algebraica para la mónada estado. Teniendo en cuenta que `MonadState` destila la esencia algebraica de una lens, se emplea el término *lens algebra* para referirse a ella (sección 3.2).
- Se generaliza la representación de una lens basada en un morfismo de mónada estado [111] en un *homomorfismo de lens algebra* (sección 3.3). Esta abstracción permite adaptar el patrón de composición para una lens algebra en el plano genérico.
- Se propone un patrón de diseño para la implementación de los repositorios y la lógica de negocio de una aplicación, utilizando lens algebras (sección 3.2.1) y lens algebras homomorphisms (sección 3.3.1) como bloques fundamentales de construcción.
- Se determina el proceso a seguir para identificar las teorías algebraicas asociadas a otras ópticas, así como las abstracciones que habilitan su composición en el nuevo plano genérico (sección 3.4).

Este capítulo utiliza el asistente de pruebas Coq para formalizar su contenido. Desafortunadamente, los tipos de datos y type classes requeridos para llevar a cabo esta tarea no estaban disponibles en una librería de programación funcional destinada a la industria. Por este motivo, fue necesario crear *Koky* [49], que se postula como un resultado indirecto pero relevante de esta tesis. Por tanto, antes de adentrarnos en las contribuciones, se ofrece una breve introducción a algunas abstracciones que forman parte de dicha librería y se adapta el ejemplo

de la universidad (sección 2.1.4) donde se aprovechará la ocasión para introducir también conceptos propios de Coq.

3.1. Koky y adaptación a Coq

A pesar del estilo funcional adoptado en Coq [106], que cuenta incluso con soporte para type classes [112], existe una notable falta de librerías funcionales destinadas a la industria[1]. Bajo esta motivación se lleva a cabo el desarrollo de Koky [49], una librería que pretende explotar los beneficios de los tipos dependientes pero manteniéndose cercana a lo que un programador de Scala o Haskell podría esperar encontrar. La librería no es ni mucho menos completa, ya que esencialmente contiene las abstracciones que eran necesarias para cubrir este capítulo, pero establece unos pilares sólidos que pueden servir de referencia para otros programadores o investigadores que se enfrenten a una situación similar. En los siguientes párrafos se mostrará la adaptación de algunas de las abstracciones que ya se vieron a lo largo del capítulo 2, junto con ejemplos para ponerlas en práctica.

La definición 1 codificaba una lens en Scala, en términos de una clase. En Coq se incluye el concepto de registro, que sirve para adaptar esta abstracción, tal y como queda recogido en Koky:

```
Record lens (S A : Type) := mkLens
{ view : S → A
; update : S → A → S
; modify (f : A → A) : S → S := λ s ⇒ update s (f (view s)) }.
Notation "ln ∼ f" := modify ln f.
```

El registro, de nombre `lens` recibe los parámetros tipo `S` y `A`, que se corresponden con el 'todo' y la 'parte' y ofrece las definiciones de `view` y `update`[2], donde el símbolo → se corresponde con el tipo función. El campo `modify` se deriva a partir de los anteriores y tiene asociada una notación infija, como refleja la última línea. Finalmente, `mkLens` no es más que el nombre asignado al constructor de instancias para este registro. Por ejemplo, la lens identidad (codificada en Scala en la figura 2.1) se adapta de la siguiente manera:

```
Definition identityLn {A} : lens A A :=
  mkLens id (λ _ ⇒ id).
```

El término **`Definition`** es uno de los nombres permitidos en Coq para introducir nuevas definiciones. El resto de la signatura indica el nombre de la definición (`identityLn`) y el parámetro tipo para la lens, que se usa en el tipo de retorno, que aparece tras el simbolo ':'. Como no podía ser de otra manera, la librería también recoge la función que establece la composición de dos lenses, para la que también se propone una notación infija:

```
Definition composeLn {S A B}
    (ln1 : lens S A) (ln2 : lens A B) : lens S B :=
  mkLens (λ s ⇒ view ln2 (view ln1 s))
         (λ s a′ ⇒ update ln1 s (update ln2 (view ln1 s) a′)).
```

[1] Merece la pena destacar la existencia de librerías con contenidos similares orientadas a una audiencia matemática, destacando quizás `https://github.com/jwiegley/category-theory`.

[2] En Coq, los diferentes registros que se muestran en un mismo scope no pueden compartir nombres. Por tanto, se reservan `get` y `put` para la definición de `MonadState` que se verá más adelante.

```
Notation "ln1 ⋙ ln2" := composeLn ln1 ln2.
```

Ahora, se podría jugar con esta composición para modificar el presupuesto de una universidad, cuyo modelo se adapta mediante registros:

```
Record department := mkDepartment
{ dpt : string
; budget : nat }.

Record university := mkUniversity
{ unv : string
; mathDep : department }.
```

Las lenses que seleccionan el presupuesto de un departamento y el departamento de matemáticas de una universidad también se adaptan como definiciones:

```
Definition budgetLn : lens department nat :=
  mkLens budget (λ d b′ ⇒ mkDepartment (dpt d) b′).

Definition mathDepLn : lens university department :=
  mkLens mathDep (λ u d′ ⇒ mkUniversity (unv u) d′).
```

Finalmente, las definiciones anteriores permiten la implementación de la lógica de negocio asociada a la universidad:

```
Definition doubleUnivBudget : university → university :=
  (mathDepLn ⋙ budgetLn) ∼ (λ b ⇒ b * 2).
```

A estas alturas, todos los elementos de esta definición deberían de estar claros.

Este capítulo utilizará los repositorios funcionales y las limitaciones que se asociaban a ellos (sección 2.2) para guiar las explicaciones. En concreto, se utilizará el repositorio funcional de la universidad, que se podría adaptar en Coq de la siguiente manera:

```
Record UniversityAlg (p : Type → Type) :=
{ getUnv : p string
; putUnv (u : string) : p unit
; modUnv (f : string → string) : p unit
; getMathDep : p department
; putMathDep (d : department) : p unit
; modMathDep (f : department → department) : p unit }.
```

El único aspecto relevante a destacar de esta definición es la notación del constructor `p`, cuyo kind es `Type → Type`, i.e. $* \mapsto *$. A partir de este repositorio sería posible adaptar la lógica que dobla el presupuesto de la universidad de manera genérica:

```
Definition doubleUnivBudget `{alg : UniversityAlg p} : p unit :=
  modMathDep (λ d ⇒ mkDepartment (dpt d) (budget d * 2))
```

La notación que rodea a `alg` en la signatura es la que utiliza Coq para recibir parámetros de manera implícita.

Como se puntualizaba al principio de la sección, Coq también ofrece soporte para type classes [112]. La adaptación de *MonadState* (definición 27) podría llevarse a cabo de la siguiente manera:

```
Class MonadState (A : Type) (m : Type → Type) `{Monad m} :=
{ get : m A
; put : A → m unit }.
```

La definición de la clase viene determinada mediante el término **`Class`** y recibe los dos parámetros esperados para esta clase. Adicionalmente, esta clase requiere

una instancia de `Monad`[3] para `m`. Con el objetivo de mostrar una instancia de esta clase, se adapta la mónada estado (definición 13):

```
Record state (S Out : Type) := mkState
{ runState : S → Out * S }.
```

La instancia de una type class se lleva a cabo a partir de la primitiva **`Instance`**, como se muestra en la siguiente definición:

```
Instance MonadState_state {S} : MonadState S (state S) :=
{ get := mkState (λ s ⇒ (s, s))
; put := λ s ⇒ mkState (λ _ ⇒ (tt, s)) }.
```

Este fragmento se corresponde con la definición `stateTMonadState` de la sección 2.3.3, para el caso particular en el que `M[_]` es la mónada *Id*.

Hasta ahora no se han explotado los beneficios de los tipos dependientes en ninguna de las abstracciones, pero resultan ser esenciales para la definición de proposiciones, donde se pueden encontrar las leyes asociadas a una clase:

```
Record MonadState_Laws {A m} `{MonadState A m} :=
{ get_get : get ≫= (λ s1 ⇒ get ≫= (λ s2 ⇒ ret (s1, s2))) =
            get ≫= (λ s ⇒ ret (s, s))
; get_put : get ≫= put = ret tt
; put_get : ∀ s, put s ≫ get = put s ≫ ret s
; put_put : ∀ s1 s2, put s1 ≫ put s2 = put s2 }.
```

A simple vista, la definición de estas leyes resulta muy similar a la que se dio en Scala. Sin embargo, hay una diferencia crucial, que hace que esta definición sea mucho más práctica. En Scala, las leyes son de tipo `Boolean` y es la implementación de la ley la que ofrece la definición de la misma. En el caso de Coq, los tipos dependientes posibilitan que las propias leyes sean definidas como un tipo particular. De hecho, la instanciación de estos tipos se corresponde con la prueba formal de que la proposición es correcta. Como consecuencia, es el compilador el que certifica que el código es correcto y no se requieren ineficientes técnicas de property-based testing [22] para ejercitar las leyes. Al final de la sección 3.2 se mostrará el entorno y la implementación de una prueba donde se pondrán en práctica estas intuiciones.

3.2. Una teoría algebraica para lens

Es inevitable encontrar ciertas similitudes entre `lens` (definición 1) y `MonadState` (definición 27). En primer lugar, existe una clara analogía entre los métodos que definen ambas abstracciones. Por un lado, nos encontramos con `view` y con `get`, que llevan a cabo una lectura a partir del estado actual. Por otro lado, `update` y `put` reflejan la actualización o reemplazo del estado actual, o de una parte de él, a partir de un valor que se recibe como argumento. En segundo lugar, ambas abstracciones ofrecen un conjunto de propiedades o leyes que deben cumplirse para que sus instancias sean consideradas válidas, que básicamente determinan cómo deben interactuar los métodos de lectura y escritura del punto anterior. En este sentido, y dejando de lado la propiedad `get_get` definida para `MonadState`, la correspondencia entre el conjunto de leyes determinado para cada abstracción permite vislumbrar una clara semejanza. Esta sección pretende formalizar la conexión existente entre `lens` y `MonadState`. Antes de esto, se procurará allanar el camino siguiendo una línea más informal.

[3] Cuya implementación no se muestra por brevedad.

Cuando se presta atención a las signaturas de los métodos `view : S → A` y `update : S → A → S` de lens, se puede observar que ambos reciben el estado actual `S` como primer parámetro. Sin embargo, el resto de elementos no parecen ser muy homogéneos entre sí. De hecho, `view` produce una salida de tipo `A`, mientras que `update` requiere una entrada adicional `A` para producir la nueva versión del estado `S`. A continuación se tratará de homogeneizar ambos métodos de manera que contemplen las nociones de entrada, salida y estado resultante, con el objetivo de identificar y abstraer las partes comunes. Se muestra el proceso con esta derivación informal:

```
  ( view   : S      → A
  , update : S → A →     S)
≃ [functional extensionality]
  ( λ s   ⇒ view s     : S      → A
  , λ s a ⇒ update s a : S → A →     S)
≃ [add contrived input to view]
  ( λ s _ ⇒ view s     : S → 1 → A
  , λ s a ⇒ update s a : S → A →     S)
≃ [add contrived resulting state to view]
  ( λ s _ ⇒ (view s, s) : S → 1 → A * S
  , λ s a ⇒ update s a  : S → A →     S)
≃ [add contrived output to update]
  ( λ s _ ⇒ (view s, s)      : S → 1 → A * S
  , λ s a ⇒ (tt, update s a) : S → A → 1 * S)
≃ [flip parameters]
  ( λ _ s ⇒ (view s, s)      : 1 → S → A * S
  , λ a s ⇒ (tt, update s a) : A → S → 1 * S)
≃ [abstract with state monad]
  ( λ _ ⇒ mkState (λ s ⇒ (view s, s))      : 1 → state S A
  , λ a ⇒ mkState (λ s ⇒ (tt, update s a)) : A → state S 1)
```

Tras el proceso de homogeneización se obtienen un par de funciones kleisli para la state monad (definición 13)), donde `S` desempeña el rol del estado en evolución. Si se va un paso más allá, se puede incluso abstraer `state S`, con lo que se llegaría a las funciones `1 → m A` y `A → m 1` para `view` y `update`, respectivamente. Estas son exactamente las signaturas que exhiben los métodos `get` y `put` de `MonadState`. Por tanto, entender `lens` como una instancia de `MonadState` —donde `A` se corresponde con el foco y `state S` actúa como efecto monádico— parece ser viable. Es importante enfatizar aquí que esta instancia de `MonadState` no sería la habitual que suele encontrarse en las librerías estándar, ya que el foco y el estado interno que evoluciona `state` no coinciden. Ahora, bajo la nueva intuición, es posible convertir una `lens` en una instancia de `MonadState` y viceversa, tal y como se muestra a continuación:

```
Instance lens_2_ms {S A} (ln : lens S A) : MonadState A (state S) :=
{ get   := mkState (λ s ⇒ (view ln s, s))
; put a := mkState (λ s ⇒ (tt, update ln s a)) }.

Definition ms_2_lens {S A} (ms : MonadState A (state S)) : lens S A :=
{| view s     := evalState get s
;  update s a := execState (put a) s |}.
```

De hecho, son estos métodos los que permiten establecer la siguiente afirmación:

Proposición 1. Existe un isomorfismo entre una instancia `MonadState A (state S)` y una very well-behaved lens `lens S A`.

Esta proposición queda formalizada en Coq bajo la siguiente definición:

```
Proposition ms_iso_lens :
  ∀ {S A} (ln : lens S A) (ms : MonadState A (state S)),
    lensLaws ln → @MonadStateLaws _ _ _ _ ms →
    ((ms_2_lens · lens_2_ms) ln = ln) ⋀ ((lens_2_ms · ms_2_lens) ms = ms).
```

Básicamente indica que para toda lens `ln` y teoría algebraica `ms`, siempre y cuando se respeten las leyes oportunas, aplicar las transformaciones de ida y vuelta debería de resultar en el valor original, en ambas direcciones. Adicionalmente, y con el objetivo de ejercitar las leyes, también se proporcionan los lemas `MonadState_state_s_induces_lens` y `lens_induces_MonadState_state_s`, que muestran cómo las transformaciones respetan las leyes cuando una abstracción se traduce en la otra. Por ejemplo, esta es la signatura de la primera de ellas:

```
Lemma MonadState_state_s_induces_lens :
  ∀ {S A : Type} (ms : MonadState A (state S)),
    @MonadStateLaws A (state S) _ _ ms → lensLaws (ms_2_lens ms).
```

Aquí se puede apreciar que si `ms` cumple las leyes de `MonadState`, entonces `ms_2_lens ms` será una very well-behaved lens.

Observación 19. Este capítulo no mostrará las implementaciones de las pruebas, ya que en general son largas, complejas y no son autocontenidas. No obstante, al final de esta sección se mostrará la implementación del lema anterior, donde también se introducirá muy brevemente el entorno utilizado, con el fin de dar una visión global del proceso de formalización. En cualquier caso, se invita al lector interesado a visitar las pruebas completas en [50], que se corresponde con el material suplementario que acompañaba a la publicación [86].

Se sostiene por tanto que `MonadState` es una generalización de `lens`, que proporciona el álgebra para acceder y manipular un estado único que está siempre disponible, pero lleva a cabo esta idea desde un nivel de abstracción más alto. Por esta razón nos referiremos a `MonadState` como *teoría algebraica de lens* o simplemente *lens algebra*, para abreviar.

Definición 28. *Lens algebra* es una forma alternativa de referirse a `MonadState`, donde se enfatiza el hecho de que es una generalización de `lens` que destila su esencia algebraica.

```
Record lensAlg (p : Type → Type) (A : Type) `{M : Monad p} :=
{ view : p A
; update : A → p unit
; modify (f : A → A) : p unit := view ≫= (update · f) }.
Notation "ln ∼ f" := modify ln f.
```

Consecuentemente, se hace referencia a un *very well-behaved lens algebra* cuando la definición anterior cumple las leyes asociadas a `MonadState`.

```
Record lensAlgLaws {p A} `{Monad p} (ln : lensAlg p A) :=
{ view_view :
    view ln ≫= (λ s1 ⇒ view ln ≫= (λ s2 ⇒ ret (s1, s2))) =
    view ln ≫= (λ s ⇒ ret (s, s))
; view_update : view ln ≫= update ln = ret tt
; update_view : ∀ s, update ln s ≫ view ln = update ln s ≫ ret s
; update_update : ∀ s1 s2, update ln s1 ≫ update ln s2 =
                           update ln s2 }.
```

Si se descarta update_update de este conjunto de leyes es posible hablar simplemente de *well-behaved lens algebra*.

Observación 20. La instancia `MonadState A (state S)` no se encuentra en las librerías oficiales de los lenguajes de programación funcionales habituales. Esto se debe en gran parte a la dependencia funcional que se impone entre el efecto y el foco de la type class, que pretende evitar ambigüedades a la hora de resolver las instancias [67]. Sin embargo, este trabajo no requiere de la resolución de instancias de forma implícita, y es por esto por lo que lens algebra se presenta mediante un registro en vez de una type class.

En el capítulo 2 se presentó a `MonadState_state` como la instancia más representativa de `MonadState`, donde el tipo de foco y el tipo del estado interno del efecto coinciden. Esta definición se corresponde con una instancia de `lens` muy particular.

Corolario 1. `MonadState_state` es isomorfa a `identityLn`[4].

Una vez precisadas las conexiones entre las abstracciones `MonadState` y `lens` —donde la primera se muestra como una versión más generalizada de la segunda— las pondremos en práctica en la implementación de una versión generalizada de la capa de datos para el ejemplo de la universidad.

Entorno e implementación de lema

Todas las pruebas que se presentan en este capítulo se han formalizado mediante el entorno *CoqIDE*. La figura 3.1 muestra el proceso de formalización del lema `MonadState_state_s_induces_lens`. El panel de la izquierda muestra la implementación de la prueba, donde el resaltado de color verde puede entenderse como la posición actual del *debugger*. El panel superior derecho indica el estado de los objetivos, es decir, muestra en que estado se encuentra la prueba tras haber ejecutado todas las instrucciones resaltadas en el panel de la izquierda. Finalmente, el panel inferior derecho muestra mensajes de información de interés, como mensajes de error.

La implementación completa del lema puede verse en la figura 3.2. Además de la signatura o cabecera que ya se introdujo con anterioridad, la implementación debe de venir acompañada con un cuerpo de la prueba, que empieza con la instrucción **`Proof.`** y que termina con el clásico **`Qed.`** (*Quod erat demonstrandum*). Si se posiciona el debugger sobre la primera de estas instrucciones, es decir, justo al principio de la prueba, se podrá observar el siguiente objetivo en el panel de la derecha, que se corresponde esencialmente con la signatura de la cabecera:

```
1 subgoal
_____________________________________(1/1)
∀ (S A : Type) (ms : MonadState A (state S)),
MonadStateLaws A (state S) → lensLaws (ms_2_lens ms)
```

La instrucción `intros.` permite extraer los antecedentes de la prueba, con el fin de acotar el objetivo de la prueba. Estos quedarán a nuestra disposición para futuros pasos hacia la consecución de la prueba:

```
1 subgoal
S : Type
A : Type
```

[4] Ambas definiciones aparecen en la sección 3.1, donde fueron adaptadas a partir de otras abstracciones dispuestas en el capítulo 2.

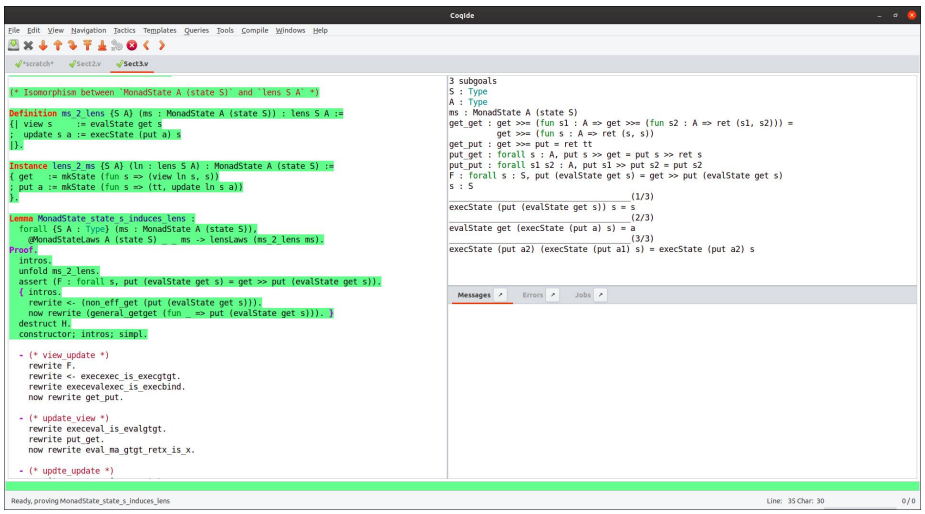

Figura 3.1: Formalización de lema en CoqIDE

```
Lemma MonadState_state_s_induces_lens :
  ∀ {S A : Type} (ms : MonadState A (state S)),
    @MonadStateLaws A (state S) _ _ ms → lensLaws (ms_2_lens ms).
Proof.
  intros.
  assert (F : ∀ s, put (evalState get s) =
                   get ≫ put (evalState get s)).
  { intros.
    rewrite ← (non_eff_get (put (evalState get s))).
    now rewrite (general_getget (λ _ ⇒ put (evalState get s))). }
  destruct H.
  constructor; intros; simpl.

  - (* view_update *)
    rewrite F.
    rewrite ← execexec_is_execgtgt.
    rewrite execevalexec_is_execbind.
    now rewrite get_put.

  - (* update_view *)
    rewrite execeval_is_evalgtgt.
    rewrite put_get.
    now rewrite eval_ma_gtgt_retx_is_x.

  - (* updte_update *)
    rewrite execexec_is_execgtgt.
    now rewrite put_put.
Qed.
```

Figura 3.2: Implementación de lema en Coq

```
ms : MonadState A (state S)
H : MonadStateLaws A (state S)
_____________________________________(1/1)
lensLaws (ms_2_lens ms)
```

Este nuevo estado nos permite ver que el objetivo de la prueba es demostrar que

se cumplen las leyes de una lens para `(ms_2_lens ms)`. Los antecedentes pasan a mostrar, entre otras cosas, que contamos con las leyes `MonadStateLaws` para la tarea. En este punto, resulta complicado establecer puntos de unión entre los antecedentes y el objetivo final, por lo que parece conveniente extraer más detalles de las definiciones. Por ejemplo, la instrucción `destruct H.` permite desenvolver las leyes específicas contenidas en `MonadStateLaws`, con lo que se llega al siguiente estado[5]:

```
1 subgoal
S : Type
A : Type
ms : MonadState A (state S)
get_get : get ≫= (λ s1 : A ⇒ get ≫= (λ s2 : A ⇒ ret (s1, s2))) =
          get ≫= (λ s : A ⇒ ret (s, s))
get_put : get ≫= put = ret tt
put_get : ∀ s : A, put s ≫ get = put s ≫ ret s
put_put : ∀ s1 s2 : A, put s1 ≫ put s2 = put s2
F : ∀ s : S, put (evalState get s) = get ≫ put (evalState get s)
_____________________________________(1/1)
lensLaws (ms_2_lens ms)
```

El siguiente paso es el de extraer los detalles de `lensLaws`, para poder conectarlos con `get_get`, `get_put`, etc. Con este objetivo se utiliza la secuencia de instrucciones `constructor; intros; simpl.`, que deriva en este estado:

```
3 subgoals
S : Type
A : Type
ms : MonadState A (state S)
get_get : get ≫= (λ s1 : A ⇒ get ≫= (λ s2 : A ⇒ ret (s1, s2))) =
          get ≫= (λ s : A ⇒ ret (s, s))
get_put : get ≫= put = ret tt
put_get : ∀ s : A, put s ≫ get = put s ≫ ret s
put_put : ∀ s1 s2 : A, put s1 ≫ put s2 = put s2
F : ∀ s : S, put (evalState get s) = get ≫ put (evalState get s)
s : S
_____________________________________(1/3)
execState (put (evalState get s)) s = s
_____________________________________(2/3)
evalState get (execState (put a) s) = a
_____________________________________(3/3)
execState (put a2) (execState (put a1) s) = execState (put a2) s
```

Algo interesante ha sucedido en este punto: en vez de tener un único objetivo, la destrucción de `lensLaws` ha provocado que ahora existan tres subobjetivos, donde cada uno se corresponde con una de las leyes que establecen el comportamiento de una very well-behaved lens. Adicionalmente, se han expandido las definiciones, por lo que ya se empiezan a encontrar términos que relacionan los antecedentes con los objetivos.

Cada uno de los guiones que aparecen en el resto de la implementación se encargará de probar uno de los subobjetivos establecidos para cada ley. El resto de explicación se centrará en el último de ellos; este es el estado de los objetivos tras posicionar el cursor sobre el tercer guión:

```
1 subgoal
S : Type
A : Type
```

[5]Se ignora `F` y la instrucción `assert` ya que no son relevantes en las explicaciones de esta sección.

```
ms : MonadState A (state S)
get_get : get ≫= (λ s1 : A ⇒ get ≫= (λ s2 : A ⇒ ret (s1, s2))) =
          get ≫= (λ s : A ⇒ ret (s, s))
get_put : get ≫= put = ret tt
put_get : ∀ s : A, put s ≫ get = put s ≫ ret s
put_put : ∀ s1 s2 : A, put s1 ≫ put s2 = put s2
F : ∀ s : S, put (evalState get s) = get ≫ put (evalState get s)
s : S
a1, a2 : A
_____________________________________(1/1)
execState (put a2) (execState (put a1) s) = execState (put a2) s
```

El objetivo está formado por una igualdad donde únicamente se muestran programas `put`. Como se puede intuir, la ley `put_put` parece ser la encargada de resolver este conflicto, pero no puede aplicarse sobre este estado. De alguna manera hay que juntar los programas `put a2` y `put a1` que se muestran en la primera parte de la igualdad. Para ello, parece necesario relacionar ejecuciones anidadas de `execState`. Afortunadamente, Coq permite la reutilización de otras proposiciones externas. En particular, es necesario este lema:

```
Lemma execexec_is_execgtgt :
    ∀ {S A B} (s : S) (st1 : state S A) (st2 : state S B),
           execState st2 (execState st1 s) = execState (st1 ≫ st2) s.
```

A grandes rasgos, indica que la ejecución de un programa sobre el resultado de la ejecución de otro programa es equivalente a ejecutar la combinación de dichos programas. Tras aplicar esta ley sobre los objetivos por medio de `rewrite execexec_is_execgtgt.` se llega al siguiente estado:

```
1 subgoal
S : Type
A : Type
ms : MonadState A (state S)
get_get : get ≫= (λ s1 : A ⇒ get ≫= (λ s2 : A ⇒ ret (s1, s2))) =
          get ≫= (λ s : A ⇒ ret (s, s))
get_put : get ≫= put = ret tt
put_get : ∀ s : A, put s ≫ get = put s ≫ ret s
put_put : ∀ s1 s2 : A, put s1 ≫ put s2 = put s2
F : ∀ s : S, put (evalState get s) = get ≫ put (evalState get s)
s : S
a1, a2 : A
_____________________________________(1/1)
execState (put a1 ≫ put a2) s = execState (put a2) s
```

El nuevo estado soporta la reescritura con `put_put`, por lo que la ejecución de la instrucción `now rewrite put_put.` nos lleva al deseado estado:

```
No more subgoals.
```

Al no existir ningún subobjetivo, es posible invocar la instrucción **`Qed`**. produciéndose el mensaje "MonadState_state_s_induces_lens is defined", que marca el fin de la prueba.

3.2.1. Diseño de capas de datos con lens algebras

Esta sección presenta la técnica y los beneficios de utilizar lens algebras para implementar la capa de datos de una aplicación. No obstante, es necesario volver al plano concreto momentáneamente para ilustrar las ideas en las que se sustentan los contenidos que se mostrarán más adelante.

Las estructuras de datos se implementan en términos de registros o alguna estructura análoga (como *case classes* en Scala), tal y como se ha podido apreciar en la codificación de la universidad y los departamentos. No obstante, aquí se presenta una codificación alternativa basada en type classes y en ópticas. Concretamente, la siguiente type class permitiría codificar una universidad:

```
Class UniversityData Univ :=
{ unvLn : lens Univ string
; mathDepLn : lens Univ department
}.
```

La clase, parametrizada con `Univ` (de kind $*$), tiene a `unvLn` y `mathDepLn` como definiciones abstractas, que se corresponden con lenses para manipular los supuestos campos subyacentes encapsulados por la representación. Dicho de otra manera, `UniversityData` clasifica a todos aquellos tipos que representan universidades. Evidentemente, el registro `university` que se mostró al principio del capítulo es una instancia de esta clase:

```
Instance UniversityData_university : UniversityData university :=
{ unvLn := mkLens name (λ u n′ ⇒ mkUniversity n′ (mathDep u))
; mathDepLn := mkLens mathDep (λ u d′ ⇒ mkUniversity (name u) d′)
}.
```

Con la definición de `UniversityData` sería posible definir la lógica que duplica el presupuesto del departamento de matemáticas de manera genérica:

```
Definition doubleUnivBudget (U : Type) `{UniversityData U} : U → U :=
  mathDepLn ∼ (λ d ⇒ mkDepartment (budget d * 2)).
```

Como se puede apreciar, la implementación trabaja para cualquier `U`, siempre y cuando se proporcione una instancia de dicho tipo para `UniversityData`. Se desconoce la existencia de literatura relacionada donde se ponga de manifiesto este patrón. Su propósito en este contexto es el de allanar el camino hacia el contenido que se mostrará en las próximas líneas.

Como se ha visto anteriormente, el concepto de lens algebra encapsula la funcionalidad que se necesita para obtener, reemplazar o actualizar una parte contextualizada en un todo. Por ello, en el contexto de un repositorio funcional, debería de ser posible sustituir el conjunto de métodos que permiten acceder a un campo en particular por esta abstracción, siguiendo el patrón propuesto en los párrafos anteriores. La abstracción resultante se muestra a continuación, donde se puede apreciar la fuerte correspondencia con respecto a `UniversityData`:

```
Record UniversityAlg p `{Monad p} :=
{ unvLn : lensAlg p string
; mathDepLn : lensAlg p department }.
```

El cambio requiere una evidencia de `Monad` para `p`, necesaria para satisfacer las dependencias del lens algebra. Ahora, es posible implementar la lógica que duplica el presupuesto de la universidad utilizando el operador estándar '$\sim$' sobre el lens algebra que apunta al departamento de matemáticas.

```
Definition doubleUnivBudget `{alg : UniversityAlg p} : p unit :=
  (mathDepLn alg) ∼ (λ d ⇒ mkDepartment (dpt d) (budget d * 2))
```

Por tanto, se han reducido considerablemente el número de métodos en `UniversityAlg`, en favor de métodos estándar que pueden ser invocados en la lógica. Esto resuelve una de las limitaciones asociadas a los repositorios funcionales que se recogían al final de la sección 2.2. No obstante, se sigue manteniendo una referencia explícita a la estructura de datos `department`. Además de la problemática

que supone esta práctica, que ya se introdujo al principio del capítulo, esta definición también muestra una clara falta de legibilidad, motivada en gran parte por la falta de composicionalidad y homogeneidad entre el código que actualiza la universidad y el que actualiza el propio departamento. Se tratarán de mitigar estas limitaciones en la próxima sección.

3.3. Componiendo lens algebras

Hasta ahora, se ha conseguido trasladar las lenses concretas a un plano más genérico, tras haber homogeneizado y abstraído `state S` de su definición, resultando en el concepto de lens algebra. A pesar de que esta nueva abstracción ofrece una interfaz muy similar a la que puede encontrarse en las lenses concretas, todavía no ha sido posible dar con una noción de composición para la misma. En esta sección se tratará de encontrarla, para lo que será necesario apoyarse en la definición alternativa de lens en términos de un morfismo de mónada estado (definición 14), que parte de unas condiciones de composición más idóneas. Adicionalmente, se aplicarán los nuevos resultados para solventar las limitaciones que surgieron en la sección 3.2.1.

Afortunadamente, la definición alternativa de lens (definición 14) (que podría adaptarse a Coq mediante `(state A ⇝ state S)`) contiene una mención explícita a `state S` en el codominio del morfismo, por lo que no se debe de llevar a cabo ningún proceso previo de homogeneización para poder abstraer dicha estructura de la definición alternativa de lens. Como resultado, se obtiene una nueva abstracción que se define a continuación.

Definición 29. `lensAlg'` es un morfismo de mónadas `state A ⇝ p` que adapta programas `state A` —que hacen evolucionar el foco al que apunta el lens algebra— en programas de tipo `p` —responsables de contextualizar y hacer evolucionar el todo.

```
Definition lensAlg' (p : Type → Type) (A : Type) `{Monad p} :=
  state A ⇝ p.
```

Tal y como el nombre de la nueva definición sugiere, esta abstracción guarda una fuerte conexión con el lens algebra. De hecho, es posible implementar una función que transforme `lensAlg'` en `lensAlg`.

```
Definition lensAlg'_2_lensAlg {p A} `{Monad p}
  (φ : lensAlg' p A) : lensAlg p A :=
{| view     := φ (mkState (λ a ⇒ (a, a)))
;  update a' := φ (mkState (λ a ⇒ (tt, a'))) |}.
```

Y viceversa.

```
Definition lensAlg_2_lensAlg' {p A} `{Monad p}
                              (ln : lensAlg p A) : lensAlg' p A :=
  mkNatTrans (λ X sax ⇒ view ln ≫= (λ a ⇒ let (x, a') := runState sax a
                                          in update ln a' ≫ ret x)).
```

De hecho, estas transformaciones sustentan el próximo lema.

Lema 1. Existe un isomorfismo entre un very well-behaved `lensAlg p A` y un morfismo de mónadas `lensAlg' p A`.

Este lema se muestra formalizado en Coq bajo la siguiente definición, de la que sólo mostramos su signatura, como viene siendo habitual.

```
Lemma lensAlg_iso_lensAlg′ :
  ∀ {p A} `{MonadLaws p} (ln : lensAlg p A) (ln′ : lensAlg′ p A),
    lensAlgLaws ln → monad_morphism ln′ →
    ((lensAlg′_2_lensAlg · lensAlg_2_lensAlg′) ln  = ln) ⋀
    ((lensAlg_2_lensAlg′ · lensAlg′_2_lensAlg) ln′ = ln′).
```

Adicionalmente, también podemos encontrar `lensAlg_induces_lensAlg′` y `lensAlg′_induces_lensAlg` en el código que acompaña a este documento, donde se muestra cómo ambas transformaciones preservan las leyes en el proceso de traducción.

A pesar de haber conseguido definir el lens algebra en términos de un morfismo, se siguen encontrando ciertas restricciones a la hora de componer dos abstracciones bajo esta nueva representación. En concreto, si se pretende componer las algebras de lens `ln1 : state S ⇝ p` y `ln2 : state A ⇝ q`, se debe imponer que `q` sea necesariamente `state S`. En general, si se quiere componer una secuencia de lens algebras `ln1 · ln2 · ... · lnN`, el único programa que se puede generalizar es aquel asociado a `ln1`, por encontrarse en la primera posición de la cadena de composición. Lamentablemente, esta es una imposición muy restrictiva que no nos permite generalizar correctamente, por lo que debe ser abordada.

Hasta el momento, se ha abstraído `state S` a partir de diferentes representaciones de ópticas para obtener abstracciones más generales. En este sentido, se puede apreciar que la nueva definición de lens algebra también contiene una referencia explícita a `state A`, que es precisamente la que impone la restricción descrita en el párrafo anterior. ¿Sería posible abstraer dicha referencia de la definición? Si se lleva a cabo un reemplazo análogo, sustituyendo `state A` por una mónada cualquiera `q` se acabaría con un simple morfismo de mónadas como resultado, con lo que resultaría inviable obtener un lens algebra a partir del mismo. No obstante, si se presta atención a `lensAlg′_2_lensAlg`, es posible apreciar cómo los programas que se usan para alimentar al morfismo, todavía sin abstraer, son exactamente los mismos que se utilizan en la instancia de `MonadState`, tal y como queda enfatizado en esta reimplementación:

```
Definition lensAlg′_2_lensAlg {p A} `{Monad p}
  (φ : lensAlg′ p A) : lensAlg p A :=
{| view      := φ get
;  update a′ := φ (put a′) |}.
```

Bajo esta premisa, es posible plantearse la abstracción de `state A`, pero no en una simple mónada, sino en una instancia de `MonadState A`, con lo que se llega a otra nueva abstracción.

Definición 30. Un *lens algebra homomorphism* abstrae la referencia a `state A` que aparece en `lensAlg′ p A`, resultando en un nuevo parámetro tipo de orden superior `q` que debe ser una instancia de `MonadState A`.

```
Definition lensAlgHom p q A `{Monad p} `{MonadState A q} :=
  q ⇝ p.
```

La nueva definición carece de referencias a `state A`, surgiendo un programa genérico `q` en su lugar. Gracias a la evidencia `MonadState A q` es posible traducir un lens algebra homomorphism en un lens algebra.

```
Definition lensAlgHom_2_lensAlg {p q A}
 `{Monad p} `{MonadState A q}
```

```
  (φ : lensAlgHom p q A) : lensAlg p A :=
{| view      := φ get
;  update a' := φ (put a') |}.
```

Lo que nos lleva a una nueva proposición.

Proposición 2. Un lens algebra homomorphism `lensAlgHom p q A` induce un lens algebra `lensAlg p A`.

Esta idea queda formalizada bajo la siguiente definición.

```
Proposition lensAlgHom_induces_lensAlg :
  ∀ {p q A} `{Monad p} `{MonadStateLaws A q} (φ : lensAlgHom p q A),
  monad_morphism φ → lensAlgLaws (lensAlgHom_2_lensAlg φ).
```

Como se puede apreciar, la traducción preserva las leyes asociadas a un very well-behaved lens (`lensAlgLaws`).

La implementación de `lensAlgHom_2_lensAlg` muestra cómo `view` se recupera a partir de `get`, mientras que `update` se genera por medio de `put`. Si se tiene en cuenta la definición 28, es posible determinar que este método no es más que un morfismo entre algebras de lens. En concreto, este método está transformando una instancia de `lensAlg q A` —o `MonadState A q`— en una instancia de `lensAlg p A`, donde el tipo de programa difiere, pero el foco sigue siendo exactamente el mismo. De hecho, la intuición detrás de esto es que el morfismo sabe traducir programas que actualizan el foco desde un determinado contexto en programas que actualizan el foco desde otro contexto más amplio que el anterior, recuperando así las nociones de *parte* y *todo*, propias de las lenses concretas.

Observación 21. Si un lens algebra es equivalente a una lens, ¿por qué no se utiliza `lensAlg q A` en lugar de `MonadState A q` en la definición 30 para enfatizar el homomorfismo? Tal y como se establecía en el Remark 20, `MonadState` suele acompañarse de una dependencia funcional entre el tipo de programa y el tipo de foco, que no era deseable para `lensAlg`. En esta ocasión sí que hay un interés por mantener esta restricción, ya que se desea mantener al programa interno `q` lo más cercano posible al foco `A`. Utilizando `MonadState` en lugar de `lensAlg` es posible beneficiarse de los mecanismos de resolución implícita provistos por el compilador para lidiar con type classes.

Una vez que se ha conseguido abstraer `state A` de la definición de `lensAlg'`, la composición de la abstracción resultante se vuelve trivial. De hecho, se corresponde con la composición de transformaciones naturales:

```
Definition composeLnAlgHom {p q r A B}
   `{MonadState B r} `{MonadState A q} `{Monad p}
    (φ : lensAlgHom p q A)
    (ψ : lensAlgHom q r B) : lensAlgHom p r B :=
  φ · ψ.
Notation "hom1 ⋙ hom2" := composeLnAlgHom hom1 hom2
```

Esta definición es closed under composition, y preserva las leyes de identidad y asociatividad[6].

De esta manera, se ha obtenido una abstracción genérica, que ofrece la interfaz de una lens y que se puede componer. Esta abstracción es más general que `lensAlg'`, tal y como puede apreciarse en la reimplementación de ésta última

[6]Es decir, esta abstracción forma una categoría [4].

en términos de la nueva abstracción, donde se concreta el programa interno `q` a `state A`:

```
Definition lensAlg' p A := lensAlgHom p (state A) A
```

A continuación, se pasarán a utilizar las nuevas definiciones y abstracciones a fin de ofrecer la versión definitiva del ejemplo de la universidad.

3.3.1. Extendiendo el diseño de capas de datos

Esta sección pretende introducir un nuevo álgebra para el departamento y utilizar las nuevas abstracciones para establecer una relación con el álgebra de la universidad. Al igual que se hizo en la sección 3.2.1, primero se planteará el mismo problema desde el plano concreto, con el fin de allanar el camino hacia la solución final.

Proporcionar una type class para codificar el departamento resulta bastante trivial. Simplemente se incluye en su interfaz abstracta una lens que permita la manipulación del presupuesto:

```
Class DepartmentData Dep :=
{ budgetLn: lens Dep nat
}.
```

El problema ahora reside en ofrecer una teoría algebraica para la universidad, ya que ésta debe de estar conectada de alguna manera con el departamento, cuya representación debería mantenerse abstracta. Para esta tarea, se propone esta definición:

```
Class UniversityData Univ :=
{ unvLn : lens Univ string
; Dep : Type
; ev : `{DepartmentData Dep}
; mathDepLn : lens Univ Dep
}.
```

Tal y como se puede apreciar, la lens `unvLn` se mantiene tal cual. También aparece `mathDepLn`, pero este incluye un cambio significativo. En vez de seleccionar `department` como parte, selecciona un tipo abstracto `Dep`, que también se incluye en la propia type class. Este tipo debe corresponderse con la representación de un departamento, es decir, debe instanciar `DepartmentData`. Esta evidencia se recoge en `ev`. El registro `department` es una instancia de `DepartmentData`:

```
Instance DepartmentData_department : DepartmentData department :=
{ budgetLn := mkLens budget (λ _ b' ⇒ mkDepartment b')
}.
```

Por su parte, el registro `university` es una instancia de `UniversityData`:

```
Instance UniversityData_university : UniversityData university :=
{ nameLn := mkLens name (λ u n' ⇒ mkUniversity n' (mathDep u))
; Dep := department
; ev := DepartmentData_department
; mathDepLn := mkLens mathDep (λ u d' ⇒ mkUniversity (name u) d')
}.
```

Aquí, la instancia fija `department` como tipo para `Dep`, con lo que puede utilizar la instancia `DepartmentData_department` como evidencia. La lógica para duplicar el presupuesto deja de ser monolítica y goza de la modularidad propia de las ópticas, donde se conecta la lens que selecciona el departamento de matemáticas con la lens que selecciona el presupuesto de un departamento:

```
Definition doubleUnivBudget (U : Type) `{UniversityData U} : U → U :=
  (mathDepLn ⋙ budgetLn ev′) ∼ (λ d ⇒ d * 2).
```

De nuevo, merece la pena destacar que no se trabaja con registros específicos, sino con cualquier `U`, siempre y cuando sea instancia de la type class `UniversityData`.

Se vuelve a mover la discusión al plano general. La primera medida a llevar a cabo es reemplazar las algebras de lens por su definición alternativa:

```
Record UniversityAlg p `{Monad p} :=
{ unvLn : lensAlg′ p string
; mathDepLn : lensAlg′ p department }.
```

Como se puede ver, simplemente se han reemplazado las referencias a `lensAlg` de la teoría algebraica producida en la sección 3.2.1 por `lensAlg′`. Veremos la ganancia obtenida por este cambio más adelante.

El siguiente paso es el de proporcionar un álgebra independiente y desacoplada para hacer evolucionar un departamento, donde se pretende abstraer la mención a la estructura de datos `department`. El resultado que se obtiene es el siguiente, donde se puede apreciar la clara correspondencia con el patrón descrito previamente en el plano concreto:

```
Record DepartmentAlg p Dep `{Monad p} :=
{ dptLn : lensAlg′ p string
; budgetLn : lensAlg′ p nat }.

Record UniversityAlg p `{Monad p} :=
{ unvLn : lensAlg′ p string
; q : Type → Type
; Dep : Type
; ev : `{DepartmentAlg q Dep}
; mathDepLn : lensAlg′ p Dep }.
```

La teoría algebraica `DepartmentAlg`, asociada a la evolución de un departamento, resulta ser muy directa, conteniendo las evidencias necesarias para acceder al identificador y al presupuesto, en términos de la definición alternativa de lens algebra. No obstante, su signatura contiene un parámetro tipo `Dep`, cuyo propósito no resulta muy claro en este punto, ya que no aparece ninguna mención al mismo desde los métodos de la definición. La intuición que ofrecemos sobre este tipo es la de una especie de puntero que contiene la información mínima sobre un departamento y nos permite acceder a él, como bien podría ser una *url* o una clave primaria. Se utiliza principalmente desde la nueva versión de `UniversityAlg`, que ha sufrido cambios importantes, y que detallamos a continuación.

En primer lugar, se puede observar cómo `department` desaparece en favor de `Dep`, un tipo miembro que nos permite abstraernos de dicha estructura de datos y que desempeña el rol de puntero descrito en el párrafo anterior. En segundo lugar, aparece otro tipo miembro `q` que representa el tipo de programas que saben hacer evolucionar un departamento, y que se utiliza desde `ev`, que no es más que una instancia de la nueva teoría algebraica `DepartmentAlg`. Esta instancia se utiliza para precisar que la universidad contiene un departamento y dispone por tanto del mecanismo para llevar a cabo la actualización. Sin embargo, nos topamos con un problema considerable: `ev` genera programas de tipo `q`, es decir, programas que evolucionan el departamento, pero nosotros estamos interesados en producir programas que hagan evolucionar la universidad.

Necesitamos una transformación natural para llevar a cabo dicha traducción y es precisamente aquí donde entran en juego los lens algebra homomorphisms, a modo de adaptador entre ambas teorías algebraicas.

```
Record DepartmentAlg p Dep `{MonadState Dep p} :=
{ dptLn : lensAlg' p string
; budgetLn : lensAlg' p nat }.

Record UniversityAlg p `{Monad p} :=
{ unvLn : lensAlg' p string
; q : Type → Type
; Dep : Type
; ev : `{DepartmentAlg q Dep}
; mathDepLn : lensAlgHom p q Dep }.
```

El principal cambio introducido ha sido en `mathDepLn`, donde se ha reemplazado un lens algebra por un homomorphism. Adicionalmente, `DepartmentAlg` establece `MonadState` como dependencia, que será necesario para poder activar la composición a este nivel. Esta aproximación aborda el resto de limitaciones de los repositorios funcionales dispuestas en la sección 2.2: se establece un patrón para definir modelos anidados y se soportan computaciones heterogéneas para cada álgebra.

Con esto, ya se dispondrían de todos los ingredientes necesarios para implementar la lógica de negocio que nos permite duplicar el presupuesto de la universidad.

```
Definition doubleUnivBudget p
   `{Monad p}
    (alg : UniversityAlg p): p unit :=
  (mathDepLn alg ⋙ budgetLn (ev alg)) ∼ (λ b ⇒ b * 2).
```

Esta definición refleja la misma elegancia que es habitual en diseños basados en ópticas. Primero, las abstracciones componen, por lo que es posible combinar el lens algebra que apunta al departamento de matemáticas con el lens algebra que apunta al presupuesto de un departamento. Segundo, el lens algebra resultante ofrece una interfaz que permite invocar el operador de actualización ($\sim$), que recibe la expresión lambda que dobla el valor de un natural. La principal diferencia que puede apreciarse es que esta definición es sin duda más general que una basada en ópticas concretas, que nos permite trabajar no sólo con estructuras de datos inmutables, sino también con otras infraestructuras basadas en estado.

Imaginemos ahora que estuviésemos interesados en implementar la lógica necesaria para conocer el presupuesto resultante tras haberlo duplicado. Hay una implementación obvia para esta tarea: primero se modifica el valor y posteriormente se recupera. El lens algebra soporta tales operaciones y el hecho de que `p` sea una mónada nos permite componer los resultados:

```
Definition doubleMathBudgetR p `{Monad p}
    (alg : UniversityAlg p) : p nat :=
  let ln := mathDepLn alg ⋙ budgetLn (ev alg)
  in ln ∼ (λ b ⇒ b * 2) ≫ view ln.
```

Sin embargo, si se tiene en cuenta el alto grado de complejidad que podría surgir en ciertas infraestructuras subyacentes, esta implementación podría no ser suficientemente óptima, ya que requiere acceder al departamento dos veces. Se podría aliviar esta situación explotando la propiedad distributiva sobre `bind` asociada a un homomorphism:

```
Definition doubleMathBudgetR′ p `{Monad p}
    (alg : UniversityAlg p) : p nat :=
  let bLn := budgetLn (ev alg)
  in (mathDepLn alg) (bLn ∼ (λ b ⇒ b ⋆ 2) ≫ view bLn).
```

Aquí, en lugar de llevar a cabo dos operaciones sencillas –que afectan a un campo en particular– del estado global, esta versión realiza una única operación más compleja sobre el foco interno y posteriormente hace un *lift* del programa resultante. Este tipo de programación es posible gracias a la adopción de transformaciones naturales, y podría resultar de utilidad en determinadas situaciones, incluso cuando la actualización afecta a varios campos internos. No obstante, adoptar este patrón hace que se diluya la perspectiva de las ópticas, por lo que la legibilidad de las definiciones disminuye. De hecho, esta última definición recuerda a la primera versión de `expertiseFl` (sección 2.1.4), donde el estilo híbrido en el que se mezclaban las ópticas con la interfaz de colecciones resultaba poco conveniente.

3.4. Identificación de nuevas optic algebras

Esta sección pretende extraer el proceso de diseño que se ha aplicado para el caso particular de una lens con el objetivo de llevarlo a otras ópticas, de manera que sea posible producir un catálogo de *optic algebras*. La figura 3.3 resume las relaciones entre lenses (primera fila) y teorías algebraicas (segunda fila) que se han descrito a lo largo de este capítulo. Todas las relaciones en el diagrama, representadas con flechas, han sido formalizadas mediante definiciones en Coq, aunque el isomorfismo entre `lens` y `lens′` ya era conocido [111].

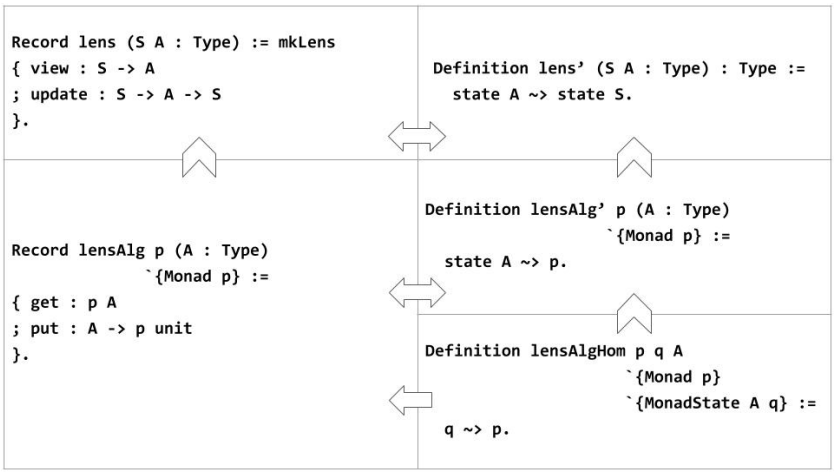

Figura 3.3: Resumen de abstracciones y relaciones

Como puede apreciarse, la columna de la izquierda relaciona la representación concreta de una very well-behaved lens y la teoría algebraica `MonadState` (o `lensAlg`). En concreto, se ha mostrado que es una relación de generalización, en el sentido en que `lens S A` puede ser definida como `MonadState A (state S)`. La columna de la derecha lidia con los aspectos de composición. En ella, se parte de una representación alternativa de lens definida en términos de un morfismo

(`state A ⇝ state S`), y se realizan varias iteraciones de abstracción hasta llegar a un lens algebra homomorphism (`lensAlgHom`), una generalización estricta que disfruta de las capacidades de composición de las que carecía `lensAlg`.

Los párrafos anteriores dan buena cuenta del proceso que se debe aplicar sobre otras ópticas para encontrar las abstracciones análogas en el plano genérico. Sin embargo, existen varios aspectos críticos que se deben contemplar:

- La conexión entre una very well-behaved `lens` y `MonadState` era bastante intuitiva. No obstante, es muy posible que haya que definir nuevas teorías algebraicas para dar soporte a otras ópticas, donde sería necesario llevar a cabo una derivación similar a la que se realizó en la sección 3.2 para homogeneizar las primitivas que definen cada óptica en particular. Esto también afecta a las leyes, que deberán diseñarse con especial atención, inspirándose en las existentes para la óptica concreta.

- Es requisito imprescindible el contar con una representación para la óptica basada en un morfismo de mónada estado sobre la que sustentar las abstracciones que habiliten la composición, como el propuesto por [111] para lens. Desafortunadamente, se desconoce la existencia de esta codificación para otras ópticas, por lo que será necesario investigar su viabilidad.

- El hecho de estar produciendo un catálogo de optic algebras deriva en una necesidad adicional: las abstracciones deben poder componerse de forma heterogénea siguiendo unas reglas análogas a las establecidas por la jerarquía de la figura 1.1 en el plano concreto.

El próximo capítulo lidia con estos retos y propone un catálogo de abstracciones experimentales que se empaquetan en una librería Scala para su posible uso industrial.

Capítulo 4

Stateless: una librería de optic algebras

Este capítulo desarrolla el **Obj. 2** (sección 1.4), llevando a la práctica los resultados obtenidos a partir del capítulo 3 mediante la implementación de una librería en Scala a la que se bautiza con el nombre de *stateless* [48]. En particular, estas son las contribuciones que se presentan en esta librería:

- Se identifican las teorías algebraicas experimentales que se corresponden con otras ópticas de la jerarquía (sección 4.1).
- Se ofrecen representaciones experimentales basadas en transformaciones naturales para dichas ópticas y se propone una noción de composición para el plano general inspirándose en ellas (sección 4.2).
- Se describen algunas operaciones y patrones recurrentes que suelen surgir cuando se utilizan optic algebras para componer consultas genéricas (sección 4.3).
- Se muestra por un lado la interpretación que permite recuperar el comportamiento habitual de las ópticas sobre estructuras de datos inmutables y por otro lado una interpretación que permite lidiar con una base de datos relacional (sección 4.4).
- Finalmente, se adapta el ejemplo de la universidad implementado en términos de la nueva librería (sección 4.5).

4.1. Teorías algebraicas inspiradas en ópticas

En primer lugar se pondrá el foco en el paquete `core.raw` donde se recopilan las teorías algebraicas inspiradas en ópticas en su versión *cruda*, es decir, representadas como type classes que contienen el conjunto de primitivas que la definen. Una de las abstracciones más relevantes en esta tesis es la de lens algebra (definición 28), cuya adaptación a Scala puede encontrarse en la figura 4.1. Como se puede apreciar, `LensAlg` recibe dos parámetros tipo `P` y `A`, que se corresponden con el efecto monádico y con el tipo de foco, respectivamente. La

```
package core.raw

trait LensAlg[P[_], A] extends MonadState[P, A] {

  /* derived methods */

  def set(a: A): P[Unit] = put(a)

  def init: P[A] = get

  def find(p: A => Boolean): P[Option[A]] =
    map(get)(a => if (p(a)) a.some else None)

  def exist(p: A => Boolean): P[Boolean] = gets(p)

  trait LensAlgLaw {

    def getGet(implicit eq: Equal[P[(A, A)]]): Boolean =
      (get ≫= (a1 => get ≫= (a2 => (a1, a2).point[P]))) ===
        (get ≫= (a => (a, a).point[P]))

    def getPut(implicit eq: Equal[P[Unit]]): Boolean =
      (get ≫= put) === ().point[P]

    def putGet(a: A)(implicit eq: Equal[P[A]]): Boolean =
      (put(a) ≫ get) === (put(a) ≫ a.point[P])

    def putPut(a1: A, a2: A)(implicit eq: Equal[P[Unit]]): Boolean =
      (put(a1) ≫ put(a2)) === put(a2)
  }

  def lensAlgLaw = new LensAlgLaw {}
}
```

Figura 4.1: Definición de lens algebra en Stateless.

abstracción, que está definida en términos de `MonadState`, está compuesta por una serie de métodos que pretenden imitar el comportamiento que ofrecen las lenses en una librería de ópticas para Scala, concretamente *Monocle* [119]. En este tipo de librerías encontramos familias de métodos que son transversales a todas las ópticas, como `find` o `exist`, cuya definición parece tener más sentido en ópticas que trabajan con múltiples focos, aunque también son viables en este escenario.

Aparentemente, el conjunto de leyes definidas en `LensAlgLaw` es muy similar al que vimos implementado con Coq en la definición original. La diferencia radica en que Coq nos permite definir las propiedades como si se tratase de tipos, gracias a su capacidad para trabajar con tipos dependientes de forma nativa, donde su instanciación es una prueba de que la propiedad se cumple. Sin embargo, las propiedades en Scala son meras igualdades que devuelven un valor `Boolean`, por lo que se suele recurrir a librerías de *property-based testing*, inspiradas en *QuickCheck* [22][1], para ejercitarlas. Esta técnica no permite formalizar pruebas, sino que trata de comprobar que la propiedad se cumple para un caso suficiente de instancias generadas automáticamente, tarea que se complica sobremanera cuando se trabaja con constructores de tipos abstractos.

En `core.raw` encontramos una gran variedad de teorías algebraicas asociadas a otras ópticas estándar. Por ejemplo, la figura 4.2 ilustra la adaptación de una teoría algebraica inspirada en *Optional*[2]. Uno de los aspectos fundamentales de esta abstracción es que no se apoya en ninguna otra teoría algebraica estándar, ya que no existe ninguna abstracción análoga que contemple la evolución de un estado que podría ser opcional. Por ello, se proponen las primitivas `getOption` y `setOption` como métodos abstractos que las instancias deben cumplimentar. El resto del módulo simplemente define combinadores derivados (algunos de ellos ya se vieron en la adaptación del lens algebra) y las leyes que definen el comportamiento correcto que se debe satisfacer por las instancias. Merece la pena destacar que, además de las abstracciones presentadas anteriormente, existen versiones tentativas de teorías algebraicas para getter, traversal, setter y fold, ópticas descritas en la sección 2.1.1[3].

La librería también incluye experimentos con teorías algebraicas inspiradas en ópticas indexadas [42], que son aquellas donde las partes seleccionadas tienen asociadas una clave o índice. La figura 4.3 muestra la teoría algebraica correspondiente a una lens indexada. En esta ocasión tampoco es posible apoyarse en otras teorías algebraicas estándar existentes, por lo que se deben definir los métodos abstractos que definen la interfaz desde cero: `get` y `set`. La principal diferencia entre estos métodos y los que ofrece `MonadState` es que el resultado de `get` viene acompañado por un valor de tipo `I`, parametrizado en el propio trait y que se corresponde con el tipo del índice. Como se puede observar, todavía no se ha trabajado en las leyes asociadas a las teorías algebraicas inspiradas en ópticas indexadas.

[1] Particularmente, *ScalaCheck* (`https://www.scalacheck.org/`) se posiciona como el principal exponente de este tipo de técnicas en el ecosistema de Scala.

[2] La óptica *Optional* es más conocida como *Affine Traversal* en el folclore [41], pero aquí se ha preferido acuñar el mismo nombre que se utiliza en Monocle por la clara influencia que esta librería tiene sobre el diseño de Stateless.

[3] Stateless también incluye una versión algebraica para *prism*, pero es una óptica extraña desde la perspectiva de la evolución de un estado, por lo que será ignorada.

```
package core.raw

trait OptionalAlg[P[_], A] extends Monad[P] { self =>

  def getOption: P[Option[A]]

  def setOption(a: A): P[Option[Unit]]

  /* derived methods */

  def modifyOption(f: A => A): P[Option[Unit]] =
    bind(getOption)(_.fold(point(Option.empty[Unit]))(setOption))

  def set(a: A): P[Unit] = void(setOption(a))

  def modify(f: A => A): P[Unit] = void(modifyOption(f))

  def isEmpty: P[Boolean] = map(getOption)(_.isEmpty)

  def nonEmpty: P[Boolean] = map(getOption)(_.nonEmpty)

  def find(p: A => Boolean): P[Option[A]] = map(getOption)(_.find(p))

  def exist(p: A => Boolean): P[Boolean] = map(getOption)(_.exists(p))

  def all(p: A => Boolean): P[Boolean] = map(getOption)(_.fold(true)(p))

  trait OptionalAlgLaw {
    implicit val _: Monad[P] = self

    def getGet(implicit eq: Equal[P[(Option[A], Option[A])]]): Boolean =
      (getOption ≫= (oa1 => getOption ≫= (oa2 => (oa1, oa2).point[P]))) ===
        (getOption ≫= (oa => (oa, oa).point[P]))

    def getPut(implicit eq: Equal[P[Option[Unit]]]): Boolean =
      (getOption ≫= (_.fold(Option.empty[Unit].point[P])(setOption))) ===
        (getOption ≫= (_.as(()).point[P]))

    def putGet(a: A)(implicit eq: Equal[P[Option[A]]]): Boolean =
      (setOption(a) ≫ getOption) === (setOption(a).map(_.as(a)))

    def putPut(a1: A, a2: A)(implicit eq: Equal[P[Option[Unit]]]): Boolean =
      (setOption(a1) ≫ setOption(a2)) === setOption(a2)
  }

  def optionalAlgLaw = new OptionalAlgLaw {}
}
```

Figura 4.2: Definición de optional algebra en Stateless.

```
package core.raw

trait ILensAlg[P[_], I, A] extends Monad[P] {

  def get: P[(I, A)]

  def set(a: A): P[Unit]

  /* derived methods */

  def modify(f: A => A): P[Unit] = bind(get) { case (_, a) => set(a) }

  def find(p: ((I, A)) => Boolean): P[Option[(I, A)]] =
    map(get)(a => if (p(a)) a.some else None)

  def exist(p: ((I, A)) => Boolean): P[Boolean] = map(get)(p)
}
```

Figura 4.3: Definición de lens algebra indexada en Stateless.

4.2. Representación natural de teorías algebraicas

En esta sección se describen las abstracciones que se localizan en el paquete `core.nat` que giran en torno a la representación de teorías algebraicas inspiradas en ópticas en términos de transformaciones naturales (introducida en la sección 3.3). La sección anterior se fundamentaba en la representación concreta de las ópticas, llevando a cabo una adaptación de las acciones de cada óptica a un plano genérico. Por su parte, esta sección se debería de apoyar en la representación de ópticas basada en transformaciones naturales (definición 14) cuya codificación en Scala se proponía de la siguiente manera:

```
type Lens[S, A] = State[A, ?] ⇝ State[S, ?]
```

Esta definición se puede leer de la siguiente manera: si hay disponible un programa que actualiza la parte y produce un valor de salida entonces es posible generar un programa que actualice el todo y devuelva dicho valor de salida. Desafortunadamente, se desconoce la existencia de representaciones basadas en transformaciones naturales para el resto de ópticas. En un experimento para acercarnos a ellas, se prestó especial atención a los siguientes candidatos para representar `Optional`s y `Traversal`s:

```
type  XOptional[S, A] = State[A, ?] ⇝ λ[x => State[S, Option[x]]]
type XTraversal[S, A] = State[A, ?] ⇝ λ[x => State[S,   List[x]]]
```

Estas definiciones resultaban ser bastante prometedoras, ya que un programa de transformación sobre una parte era capaz de generar un programa que llevase a cabo dicha transformación sobre cada uno de las partes contextualizadas en el todo, y producir un valor de salida asociado a cada una de ellas. El uso de `Option` o `List` en la salida queda por tanto determinado por el tipo de óptica. Resulta igualmente satisfactorio el hecho de que una `Lens` también pueda encajar en esta estructura, utilizando `Id` como envoltorio para el tipo de salida:

```
type Lens[S, A] = State[A, ?] ⇝ λ[x => State[S, Id[x]]]
```

```
package core.nat

trait OpticAlg[P[_], A, Ev[M[_], _] <: Monad[M], F[_]] extends Monad[P] {

  type Q[_]

  implicit val ev: Ev[Q, A]

  implicit val fev: Functor[F]

  val hom: Q ⇝ λ[x => P[F[x]]]
}
```

Figura 4.4: Abstracción unificadora de optic algebras.

El resto de abstracciones que se verán en esta sección se basa en estas intuiciones, aunque es importante destacar que no se han formalizado.

Observación 22. Este trabajo de investigación no postula estas representaciones como ópticas válidas: por un lado, no se ha encontrado el repertorio de leyes que nos permita implementar un isomorfismo entre un very well-behaved optional y `XOptional`; por otro lado se ha descartado la posibilidad de encontrar un isomorfismo entre un traversal y `XTraversal`, ya que éste último se corresponde con una abstracción más débil, en la que la actualización de la parte depende del valor actual de la misma y en ningún caso se podría definir un método derivado que asigne un valor arbitrario y único a cada parte[4].

Teniendo en cuenta la estructura común que existía entre las diversas definiciones, se propone una definición de álgebra de óptica donde se lleva esta idea al plano genérico, que puede encontrarse en la figura 4.4. El elemento principal de esta abstracción es `hom`, abreviación de homomorphism, que se corresponde con una transformación natural `Q ⇝ λ[x => P[F[x]]]`. Como se puede apreciar, `Q` abstrae `State[A, ?]`, `P` abstrae `State[S, ?]` y `F` se corresponde con `Id`, `Option` o `List`, es decir, el constructor que envuelve al tipo de salida. Otro elemento interesante de esta definición es `ev` que restringe las capacidades del programa origen `Q`.

De esta manera, seleccionando `MonadState` como restricción para `Q` y seleccionando `Id` como envoltorio para `F`, se recupera un lens algebra homomorphism (definición 30). En particular, Stateless define esta abstracción mediante herencia, tal y como puede apreciarse en la figura 4.5, donde se proporcionan tales argumentos a `OpticAlg`. De manera adicional, la abstracción también extiende `core.raw.LensAlg`, ofreciendo una implementación para `get` y `put` en términos de `hom`, donde se explota la proposición 2. También puede apreciarse cómo la composición con otros homomorphisms codificada en `composeLens` se corresponde prácticamente con la composición de transformaciones naturales mediante `compose`, aspecto que quedaba plasmado en los últimos párrafos de la sección 3.3. Se invita al lector a indagar por el resto del catálogo de combinadores de esta abstracción, del que sólo se han mostrado unos pocos por concisión. Por último,

[4] Aunque esta funcionalidad no se explota en Monocle, la abstracción tiene la capacidad para llevarla a cabo. De hecho, la representación de ópticas basada en comonad coalgebras nos permitió realizar una contribución en dicha librería para solucionar un problema similar en la construcción de traversals (`https://github.com/julien-truffaut/Monocle/pull/502`).

```
package core.nat

trait LensAlg[P[_], A] extends OpticAlg[P, A, MonadState, Id]
    with raw.LensAlg[P, A] {

  override def get: P[A] = hom[A](ev.get)

  override def put(a: A): P[Unit] = hom(ev.put(a))

  def composeLens[B](ln: LensAlg[Q, B]): LensAlg.Aux[P, ln.Q, B] =
    LensAlg(hom compose ln.hom)(this, ln.ev)

  ...

  trait NatLensAlgLaw extends LensAlgLaw {

    def hom1[X](x: X)(implicit eq: Equal[P[X]]): Boolean =
      hom(x.point[Q]) === x.point[P]

    def hom2[X, Y](qx: Q[X])(f: X => Q[Y])(implicit eq: Equal[P[Y]]): Boolean =
      hom(qx >>= f) === (hom(qx) >>= (f andThen hom))
  }

  def natLensAlgLaw = new NatLensAlgLaw {}
}

object LensAlg {

  type Aux[P[_], Q2[_], A] = LensAlg[P, A] { type Q[x] = Q2[x] }

  def apply[P[_], Q2[_], A](
      hom2: Q2 ~> P)(implicit
      ev0: Monad[P],
      ev1: MonadState[Q2, A]): Aux[P, Q2, A] =
    ...
}
```

Figura 4.5: Definición de lens algebra homomorphism

destacamos las leyes asociadas a esta estructura, que simplemente establecen que `hom` debe tratarse de un morfismo de mónadas (definición 12).

En la figura 4.4 se ignoró el hecho de que `Q` no se corresponde con un parámetro tipo, sino que aparece definido en términos de un *type member*. Este mecanismo de Scala podría entenderse vagamente como una variante de las denominadas *type families* [15] de Haskell. Aunque en teoría un type member y un parámetro tipo se pueden entender como sinónimos, en la práctica la situación resulta bien diferente. En este caso particular se ha optado por utilizar la primera opción, por la ganancia que se obtiene a la hora de inferir tipos cuando se realiza composición de homomorphisms. El tipo `Aux` que se define en el companion object de `LensAlg` surge para dar soporte a esta decisión, y suele ser un patrón muy habitual en Scala, especialmente cuando se trabaja con librerías de programación genérica como *Shapeless* [45]. La librería explota este tipo en la práctica totalidad de los combinadores, a fin de evitar perder la información interna asociada a los homomorphisms.

En `core.nat` pueden encontrarse otras teorías algebraicas que extienden de `OpticAlg`. Por ejemplo, existe el módulo `TraversalAlg` que se correspondería con un traversal algebra homomorphism, del que simplemente mostramos su signatura:

```
trait TraversalAlg[P[_], A] extends OpticAlg[P, A, MonadState, List]
    with raw.TraversalAlg[P, A]
```

Como se puede apreciar, la única diferencia con respecto a la signatura de `LensAlg` es que `TraversalAlg` configura `List` como tipo de constructor para envolver la salida. La elección del tipo de constraint también es fundamental a la hora de caracterizar este tipo de homomorphism, tal y como mostramos en la siguiente signatura, que se corresponde con la de un fold algebra homomorphism:

```
trait FoldAlg[P[_], A] extends OpticAlg[P, A, MonadReader, List]
    with raw.FoldAlg[P, A]
```

El único cambio con respecto a la signatura anterior reside en que se utiliza `MonadReader` en lugar de `MonadState` con lo que se elimina la posibilidad de que el programa `Q` pueda llevar a cabo operaciones de actualización, convirtiéndose por tanto en un álgebra de sólo lectura. En general, este paquete incluye una representación natural asociada a cada uno de los módulos definidos en `core.raw`, incluyendo la variante indexada. En este sentido, la librería también ofrece un módulo experimental `IOpticAlg` que pretende unificar todos los homomorphisms de álgebras indexadas. La única diferencia con respecto a `OpticAlg` reside en la definición de su método `hom`:

```
val hom: λ[x => I => Q[x]] ⇝ λ[x => P[F[x]]]
```

Aquí puede apreciarse que la parte izquierda de la transformación natural se ha visto alterada, consistiendo ahora en una función que genera un programa de tipo `Q` tras recibir el índice correspondiente.

4.3. Operaciones y estructuras comunes

Las librerías de ópticas han identificado determinados complementos que suelen ser habituales cuando se llevan a cabo diseños que giran en torno a estas abstracciones. Por ejemplo, una situación habitual es la de tener que lidiar con

mapas de clave-valor. A primera vista, un traversal indexado (donde el mapa en sí se corresponde con el todo y el tipo del valor se corresponde con la parte) parece la abstracción idónea para llevar a cabo lecturas y actualizaciones sobre el contenido de dicha estructura. Sin embargo, hay que tener en cuenta que un traversal definido de esta manera no tiene capacidad para añadir nuevas claves. Por ello, librerías como Monocle proporcionan una type class `At`[5] para suplir esta carencia:

```
abstract class At[S, I, A] extends Serializable {
  def at(i: I): Lens[S, A]
}
```

Como se puede ver, esta clase identifica el tipo de estructuras para las que es posible generar una lens a partir de un índice. La siguiente instancia de `At` permite abordar el caso particular del mapa:

```
implicit def atMap[K, V]: At[Map[K, V], K, Option[V]] = At { i =>
  Lens(_.get(i))(optV => map => optV.fold(map - i)(v => map + (i → v)))
}
```

Merece la pena destacar que esta instancia define `Option[V]` como el tipo de parte seleccionada, donde `None` se corresponde con el valor asociado a una clave que no está definida en el mapa.

Entre las aplicaciones de ejemplo que se utilizaron para validar Stateless también surgió esta problemática. Por ello, se llevó a cabo el experimento de llevar `At` al plano genérico, que resultó en la siguiente definición que puede encontrarse en el paquete `core.nat.op`:

```
trait At[P[_], I, A] {
  type Q[_]
  def at(i: I): LensAlg.Aux[P, Q, Option[A]]
}
```

En este caso, dado un índice, es posible generar un lens algebra homomorphism, cuyo foco está prefijado a ser opcional, a diferencia de la definición para Monocle. Stateless también adapta `FilterIndex`[6], una clase que genera traversals que únicamente selecciona aquellas partes que satisfacen un predicado sobre el índice dado como entrada:

```
trait FilterIndex[P[_], I, A] {
  type Q[_]
  def apply(p: I => Boolean): ITraversalAlg.Aux[P, Q, I :: HNil, A]
}
```

La principal diferencia con respecto a la versión de Monocle, dejando de lado las consecuencias obvias de llevar la abstracción al plano genérico, es que aquí se devuelve una óptica indexada, familia de ópticas que no está contemplada en Monocle.

Con el objetivo de acercar todas estas ideas a una audiencia industrial, donde términos como *traversal algebra homomorphism* podrían resultar cuanto menos poco atractivos, se llevaron a cabo varios experimentos para empaquetar estos aspectos en abstracciones más extendidas. Los resultados pueden encontrarse en

[5] https://github.com/julien-truffaut/Monocle/blob/master/core/shared/src/main/scala/monocle/function/At.scala

[6] https://github.com/julien-truffaut/Monocle/blob/master/core/shared/src/main/scala/monocle/function/FilterIndex.scala

```
package core.nat.lib

trait MapAlg[P[_], K, V] {

  type Q[_]

  val atEv: At[P, K, V]

  val filterIndexEv: FilterIndex.Aux[P, Q, K, V]

  implicit val M: Monad[P]

  /* derived methods */

  def apply(k: K): LensAlg[P, Option[V]] = atEv.at(k)

  def get(k: K): P[Option[V]] = apply(k).get

  def update(k: K, v: V): P[Unit] = apply(k).set(Some(v))

  def remove(k: K): P[Unit] = apply(k).set(None)

  def keys: P[List[I]] =
    filterIndexEv.filterIndex(const(true)).getList.map(_.map(_._1))

  ...
}
```

Figura 4.6: Definición de mapa genérico en términos de homomorfismos.

el paquete `core.nat.lib`, donde destacamos `MapAlg` (figura 4.6). Esta abstracción tenía como objetivo principal el de presentar al programador una interfaz cercana a la que podría encontrarse en la librería estándar de Scala para lidiar con mapas, pero con las ventajas de encontrarse en el plano general. Como se puede observar, los métodos abstractos que definen la clase requieren evidencias de las utilidades `At` y `FilterIndex` presentadas en los párrafos anteriores. A partir de ellas se podría derivar funcionalidad para consultar (`get`), actualizar (`update`) o borrar (`remove`) elementos del mapa, así como consultar todas las claves existentes (`keys`), entre otros.

4.4. Interpretaciones del lenguaje

En esta sección se describen brevemente las interpretaciones soportadas por Stateless. En primer lugar, todas las teorías algebraicas y utilidades descritas a lo largo de esta sección disponen de instancias para `State`, que se ubican en el paquete `smonocle`, de lo que se infiere que las instancias dependen de Monocle. Por ejemplo, la figura 4.7 muestra la instancia de `StateT` para `LensAlg`, que requiere la existencia de una lens de Monocle (renombrada en este ámbito como `MLens`) para su implementación. Tiempo después de introducir esta definición en Stateless, se descubrió que ya se había contemplado anteriormente por Abou-Saleh *et al.* en [1, Definition 4.1], donde básicamente se generaliza la intuición de Shkaravska [111] que ya se presentó en la definición 14.

Stateless incluye otra interpretación muy experimental sobre bases de datos

```
implicit def asLensAlg[F[_]: Monad, S, A](
    ln: MLens[S, A]): LensAlg.Aux[StateT[F, S, ?], StateT[F, A, ?], A] =
  LensAlg[StateT[F, S, ?], StateT[F, A, ?], A](
    λ[StateT[F, A, ?] ~> StateT[F, S, ?]] { sa =>
      StateT(s => sa.xmap(ln.set(_)(s))(ln.get)(s))
    })
```

Figura 4.7: Instancia de `LensAlg` para `StateT`.

```
trait UniversityAlg[P[_], U] {
  val unv: LensAlg[P, String]
  type Q[_]
  type D
  val Department: DepartmentAlg[Q, D]
  val mathDep: LensAlg.Aux[P, Q, D]
}

trait DepartmentAlg[P[_], D] {
  val dpt: LensAlg[P, String]
  val budget: LensAlg[P, Int]
}

def doubleUnivBudget[P[_], U](implicit alg: UniversityAlg[P, U]): P[Unit] = {
  import alg._, Department._
  (mathDep composeLens budget).modify(_ * 2)
}
```

Figura 4.8: Adaptación del ejemplo de la universidad en Stateless

relacionales, utilizando la librería Doobie [94] como intermediaria. Esta interpretación, que puede encontrarse en el paquete `doobie`, tiene asociadas muchas limitaciones destacando la generación de una avalancha de queries [44]. De hecho, este problema fue uno de los desencadenantes de nuestro aterrizaje en el campo de LINQ, que más tarde abriría la puerta a toda la investigación que se presenta en la parte III de esta tesis y que derivó en la paralización del desarrollo de esta librería hasta la fecha.

4.5. Adaptación del ejemplo de la universidad

Una vez que conocemos cómo se codifican las diferentes abstracciones en Scala, es posible adaptar el ejemplo de la universidad, que queda recogido en la figura 4.8. En esta implementación particular, se reutiliza `LensAlg` tanto para los campos sencillos como para los que conectan álgebras, aunque en este último caso es necesario utilizar la variante `Aux` para reflejar la conexión entre `Q` y el programa interno del homomorphism. En cuanto a la lógica, tampoco existe una diferencia notable. El cambio más relevante es la importación de las definiciones de `alg` para simplificar la expresión que produce el programa resultado.

A continuación, se proporciona la instancia del álgebra que permite evolucionar estructuras de datos inmutables, para lo que nos apoyaremos en la mónada estado y en ciertas utilidades ofrecidas por Monocle. Primero se definirán las estructuras de datos que se desean evolucionar:

```
implicit object StateUniversityAlg
    extends UniversityAlg[State[SUniversity, ?], SUniversity] {
  val unv = SUniversity.unv
  type Q[x] = State[SDepartment, x]
  type D = SDepartment
  val Department = new DepartmentAlg[State[SDepartment, ?], SDepartment] {
    val dpt = SDepartment.dpt
    val budget = SDepartment.budget
  }
  val mathDep = SUniversity.mathDep
}
```

Figura 4.9: Instancia de `UniversityAlg` para `State`.

```
@Lenses case class SUniversity(unv: String, mathDep: SDepartment)
@Lenses case class SDepartment(dpt: String, budget: Int)
```

La anotación `@Lenses` es una *macro annotation* [12] que genera lenses concretas automáticamente para cada campo y las recopila en el companion object asociado a la clase.

La instancia de `UniversityAlg`, que se presenta en la figura 4.9, utiliza `State[SUniversity, ?]` como el tipo de programa que hace evolucionar la universidad. Por su parte, el programa interno `Q` se define en términos de `State[SDepartment, ?]`, programa que hace evolucionar el departamento, y que se utiliza para instanciar el álgebra interno de tipo `Department`. Finalmente, `unv`, `dpt`, `budget` y `mathDep` se generan a partir de las ópticas correspondientes generadas por Monocle, utilizando `asLensAlg` como conversor, que el compilador invocará de manera implícita. Con la instancia ya definida, podemos generar programas que evolucionen el estado de la universidad:

```
val urjc = SUniversity("urjc", SDepartment("math", 10000))
doubleUnivBudget.exec(urjc)
// res: SUniversity(urjc,SDepartment(math,20000))
```

Primero se define el estado inicial `urjc` que se quiere hacer evolucionar. Después, se genera el programa que duplica su presupuesto mediante `doubleUnivBudget`, donde el álgebra también se resuelve de forma implícita. El último paso consiste en ejecutar el programa resultante contra el estado inicial mediante `exec`. La línea comentada indica la salida resultante, donde se refleja que el presupuesto del departamento de matemáticas ha sido duplicado. El fichero *README* de Stateless [48] contiene una sección *Getting Started* donde se muestra otro ejemplo sencillo en el que existen más niveles de anidamiento.

Capítulo 5

Discusión

Este capítulo discute los resultados obtenidos a lo largo de la parte II de esta tesis. La estructura de este capítulo se refleja en los siguientes puntos:

- Se ofrece una comparativa entre las optic algebras y otras abstracciones del estado del arte que mezclan ópticas con efectos (sección 5.1).
- Se muestra cómo la aproximación aborda los problemas asociados a los repositorios funcionales, donde también se enfatizan los matices que diferencian las optic algebras de MTL (sección 5.2).
- Se describen las limitaciones de la aproximación que hacen que su uso industrial no resulte viable (sección 5.3).

5.1. Optic algebras y abstracciones relacionadas

Como ya se pudo ver en el capítulo 2, es posible encontrar trabajos donde se relacionan ópticas y efectos en la literatura. En esta sección, se tratará de comparar tales resultados con las abstracciones introducidas a lo largo de esta sección. Concretamente, se centrará en profunctor lenses (definición 23), monadic lenses (definición 24) y monadic bxs (definición 25).

5.1.1. Profunctor Lenses

En primer lugar, se recordará la definición de una profunctor lens:

```
Definition pLens S T A B := ∀ p `{Cartesian p}, p A B → p S T.
```

Con la nueva perspectiva adquirida a lo largo de este capítulo, es posible apreciar que esta abstracción también trabaja con programas genéricos de tipo p, que en vez de ser monádicos, resultan ser profuntores. Sin embargo, existe un matiz importante que hace que no sea posible utilizar esta abstracción para el tipo de efectos que deseamos, y es el hecho de que la signatura indique que la función es capaz de trabajar con cualquier profuntor cartesiano, tal y como se describe en el siguiente ejemplo.

Imaginemos que tenemos una profunctor lens apuntando al presupuesto de un departamento, cuyo tipo es `pLens Dep Dep nat nat`, donde `Dep` se corresponde con el índice que permite acceder a la información asociada al departamento.

Entre las posibles instancias de profuntores cartesianos se encuentra *UpStar*, tal y como muestran Pickering *et al* en [105]. Si se elige *Constant* en combinación con *UpStar* se estará recuperando el método `view` propio de las lenses concretas. De este modo, se obtiene una función pura de tipo `Dep → nat`. Esto entra directamente en conflicto con que `Dep` sea un índice, que impone una computación con efectos como el único mecanismo posible para recuperar el presupuesto del departamento.

Como ya se había postulado con anterioridad, una profunctor lens no es más que una representación alternativa de una lens concreta, y a pesar de la elegancia que ofrece en términos de composición y al nuevo universo de profunctores que se abre para identificar nuevos métodos estándar, también es generalizada por un lens algebra (siempre y cuando nos restrinjamos a su versión monomórphica). Si se pretende relacionar lens algebras con otras abstracciones, éstas deben de contemplar los efectos computacionales en su definición de una manera más directa.

5.1.2. Monadic Lenses

Las monadic lenses, a pesar de incluir un efecto monádico en su declaración, tampoco dan soporte al tipo de infraestructuras basadas en estado en las que estamos interesados. En este sentido, la ausencia de efectos en `mview` es la principal evidencia de esto, en línea con lo que ya se contó al comparar lens algebras con profunctor lenses. A grandes rasgos, podemos decir que las lens algebras y las monadic lenses persiguen objetivos diferentes. Por un lado, las lens algebras pretenden abstraerse de las estructuras inmutables de datos mediante la parametrización de un efecto que nos sirve para manipular y acceder al estado. Por otro lado, las monadic lenses pretenden enriquecer las operaciones de las lenses concretas con efectos tales como la parcialidad o el logging, pero en la tarea no abandonan las estructuras inmutables de datos. Más tarde, en la sección 5.1.3, analizaremos si las lens algebras dan soporte a este tipo de efectos o no.

En esta comparación, se han encontrado observaciones muy relevantes a partir de la discusión que propone Abou-Saleh *et al.* sobre la importancia de mantener `mview` libre de efectos. En particular, hay una pregunta que surge de manera natural: ¿es seguro definir `view` con efectos tal y como hacemos en la definición de lens algebra? Para contestar a esta pregunta de forma más precisa, resulta conveniente introducir una consecuencia derivada de sus leyes:

Lema 2. Considera una very well-behaved lens algebra `ln` cuyo tipo es `lensAlg p A`. La siguiente propiedad se deriva a partir de sus leyes:

```
∀ (X : Type) (px : p X), view ln ≫ px = px.
```

Este lema muestra cómo una invocación a `view`, donde su salida es ignorada, es redundante. Sin embargo, si `view` lleva a cabo algún efecto, parece complicado reconciliar esta idea. Por ejemplo, un programa que ejecuta una consulta sobre una base de datos y después ignora el resultado, no es igual a un programa donde simplemente se devuelve el resultado. Por esta razón, tenemos que relajar la noción de igualdad asociada a nuestras leyes, enfatizando que nos referimos a igualdad de estado resultante. Para el ejemplo anterior, consideramos que dos programas son equivalentes si ambos devuelven el mismo resultado y si ambos producen el mismo estado final en la base de datos. Por último, nos

gustaría mencionar que es posible encontrar una variante de este lema para el transformador de mónada de estado en [1, Lemma 2.7.]

5.1.3. Entangled State Monads

Sin duda alguna, existe una gran similitud entre una entangled state monad y nuestro trabajo. En primer lugar, Abou-Saleh *et al.* recuperan una very well-behaved lens `lens S A` a partir de la instancia `BX (state S) S A`, que ofrece una interfaz no sólo para acceder a la parte (`A`), sino también al todo (`S`). De hecho, aquí se sigue una aproximación muy similar a la que vimos en Proposition 1, donde también se utiliza la state monad para recuperar una lens. En segundo lugar, `BX` también está muy relacionada con la forma en la que codificamos las capas de datos. En particular, la restricción `id` que debe proporcionarse junto con los repositorios se corresponde con los métodos `getL` y `putL`, aunque es ignorada, ya que no estamos interesados en evolucionar el todo directamente (al menos, en los ejemplos que se muestran en este capítulo). Por su parte, `getR` y `putR` se corresponden con un lens algebra en la capa de datos, como podría ser `budgetLn`.

Entonces, ¿no es posible utilizar `BX` para programar capas de datos? En primer lugar, la composición de dicha abstracción está definida para un well-behaved BX [1, Definition 3.8.], donde las operaciones `getL` y `getR` son esencialmente puras. Esto va en contra de la filosofía de nuestros casos de uso más relevantes, donde sí que se requieren efectos, por ejemplo, para realizar una lectura sobre una base de datos. Aquí, vuelve a entrar en juego la discusión mantenida al final de la sección 5.1.2, donde se hablaba de equivalencia en términos de igualdad de estado resultante. En este sentido, el método `get` de un lens algebra puede contener efectos para acceder a los datos, pero no debería alterar el estado resultante de la aplicación. En segundo lugar, nuestro trabajo mantiene un fuerte interés en desplegar efectos computacionales para repositorios anidados, que se consigue mediante la introducción de homomorphisms de lens algebras, mientras que `BX` no ofrece soporte para este aspecto.

Merece la pena destacar un aspecto muy importante que se deriva de las entangled state monads, que afecta a la sobrescritura (*overwritability*), que responde a la pregunta sobre si las lens algebras soportan efectos adicionales, como la parcialidad o el logging. Este aspecto está fuertemente ligado a la controvertida ley `update_update` en el contexto de las lens algebras. En este sentido, una very well-behaved lens algebra hereda las mismas limitaciones y por tanto no los soporta. De hecho, si se pretende enriquecer las instancias con este tipo de efectos, debemos adoptar la noción de well-behaved lens algebra, donde esta ley quedaría descartada.

5.2. Optic algebras, repositorios y MTL

Las optic algebras resuelven los problemas asociados a los repositorios que se ilustraban en el capítulo que introducía esta tesis. En primer lugar, se han propuesto abstracciones estándar que permiten recopilar operaciones habituales que suelen encontrarse en los repositorios para lidiar con el estado de una aplicación. Por ejemplo, la funcionalidad que se requiere para consultar, reemplazar o modificar un campo en particular del modelo queda encapsulada bajo

un lens algebra, lo que reduce notablemente el número de operaciones del repositorio. Más allá de las álgebras inspiradas en lenses, Stateless ofrece un amplio catálogo que permite lidiar con campos opcionales (optional algebra), campos multivaluados (traversal algebra), campos de los que sólo se requiere lectura (álgebras inspiradas en read-only optics) o incluso las variantes indexadas que incluyen una clave que acompaña a las partes seleccionadas. De esta manera, los repositorios se enriquecen enormemente gracias a la riqueza de abstracciones proporcionada por la jerarquía de ópticas. En este sentido, también merece la pena destacar los empaquetados de ópticas soportados por la librería, por ejemplo mediante la combinación de un traversal indexado con una factoría de lenses para poder simular el comportamiento de un mapa en el plano genérico. Este tipo de interfaces incluso se podrían explotar por programadores que sean ajenos a las abstracciones y patrones de las ópticas.

El otro problema evidente que surge a la hora de implementar repositorios para acceder a la capa de datos de una aplicación es el de la falta de modularidad, donde un repositorio monolítico suele desplegar todas las operaciones de acceso asociadas a las diversas entidades. Tal y como se mostraba en el ejemplo de la universidad, esto derivaba en una importante falta de granularidad en la que la lógica que duplicaba el presupuesto de un departamento requería cargar toda la información contenida por éste, lo que podría llegar a resultar inviable o poco eficiente. De forma alternativa, se podrían introducir métodos de grano más fino para lidiar con determinados campos del estado, pero esto derivaría en métodos complejos, que deberían contener toda la información necesaria que permitiese la identificación del campo concreto al que brindar acceso. Estas limitaciones se solventan con los homomorphisms de optic algebras, que fomentan la modularidad permitiendo la separación de las álgebras asociadas a cada entidad. Esta práctica permite que cada álgebra se encargue de brindar acceso a los campos de una entidad particular con métodos estándar que no requieren toda la información sobre el contexto, ya que esta tarea será responsabilidad del homomorphism. Para el caso particular en que un álgebra apunte a otra entidad, se delega en el subálgebra correspondiente.

Éste último párrafo refleja un aspecto diferenciador importante que surge entre Stateless y la aproximación MTL en general, que pasamos a describir en las siguientes líneas. Por un lado, la lógica de negocio implementada en MTL gira en torno a un único tipo de computación (`P[_]`) que debe instanciar todas las teorías algebraicas de las que dependa dicha lógica (`MonadState`, `MonadError`, etc.), resultando en un código muy elegante. Sin embargo, proporcionar una instancia concreta para `P` podría no resultar trivial, ya que la combinación de efectos deriva en la necesidad de transformadores de mónadas [80, 40], que son, esencialmente, morfismos de mónadas. Por otro lado, la aproximación de Stateless contempla la heterogeneidad de efectos, proporcionando un tipo de computación diferente (`P[_]`, `Q[_]`, etc.) para cada álgebra anidada. De hecho, en vez de mantener la lógica de negocio ajena a dicha complejidad, Stateless la hace explícita, enmascarando los morfimos de mónadas en forma de composición de optic algebras. Es precisamente esta solución intermedia la que habilita la posibilidad de describir álgebras anidadas.

5.3. Limitaciones de la aproximación

La aproximación basada en optic algebras no está exenta de limitaciones, tal y como se expone en los siguientes puntos:

- A pesar de la modularización soportada gracias a los homomorphisms de optic algebras, la abstracción introducida por las álgebras anidadas es *leaky*, ya que contaminan el modelo haciendo explícita la heterogeneidad subyacente. En particular, para cada entidad anidada, se requiere la introducción de un tipo miembro abstracto, una instancia del subálgebra involucrada y un homomorphism de optic algebra. Esto resulta inviable para una librería enfocada a la industria, tal y como se infiere a partir de un comentario de Reddit donde se critica el enfoque[1].

- Las abstracciones de Stateless que se inspiran en otras ópticas que no sean lenses no están formalizadas. De hecho, no se han encontrado representaciones para otras ópticas basadas en morfismos de la mónada estado, por lo que ni siquiera existe un punto de partida sobre el que trabajar los aspectos de composición. Las abstracciones dispuestas se basan en la intuición personal y algunas de ellas ya han demostrado no tener todo el potencial que exhibe su análoga en el plano concreto, por ejemplo, en el caso de un traversal algebra, que no permite la configuración de un valor específico para cada parte seleccionada.

- El hecho de que las nuevas abstracciones que trabajan en el plano genérico se inspiren en ópticas concretas no garantiza que puedan explotar todo el potencial de su versión análoga. Esto se puede vislumbrar con el homomorphism de getter algebra, donde la implementación del combinador `fork` no es viable. Por tanto, el proceso de generalización deriva en una perdida de expresividad.

- A pesar de la generalización introducida por las optic algebras, la aproximación, que se basa en MTL, asume la existencia de efectos computacionales. Este hecho limita la declaratividad y reduce las oportunidades para introducir optimizaciones. Esto se aprecia de forma muy acusada en la interpretación a bases de datos relacionales, donde las consultas SQL generadas son subóptimas, requieren una consulta adicional por cada nivel de anidamiento introducido en el modelo.

Afortunadamente, estas limitaciones se abordan en Optica, tal y como se mostrará más adelante.

[1] `https://www.reddit.com/r/scala/comments/7swn90/lens_state_is_your_father_and_i_can_prove_it/`

Parte III

Ópticas y Lenguajes

La tercera parte de este documento se centra en los objetivos 3 y 4 dispuestos en la sección 1.4. El capítulo 6 desarrolla el primero de ellos, para lo que se apoya en la práctica totalidad de los contenidos publicados en [87]. Concretamente, se diseña un lenguaje que recoge la esencia de un subconjunto de ópticas con el que es posible definir consultas genéricas, se propone su semántica estándar y se definen otras semánticas no estándar para otras fuentes de datos (SQL, XQuery y T-LINQ) que posibilitan traducciones eficientes a consultas específicas. El capítulo 7 expande los contenidos de [87, Sect. 7], detallando no sólo la implementación en Scala del lenguaje y su semántica estándar, sino también las interpretaciones particulares para otras fuentes de datos. Finalmente, el capítulo 8 discute los resultados derivados de ambos capítulos.

Esta parte reutiliza muchos de los combinadores dispuestos en la parte I de este documento y pone en práctica la aproximación tagless-final para la implementación de S-Optica. Las limitaciones que surgen en la aproximación descrita en la parte II (complejidad accidental en el modelo de datos, imposibilidad para definir determinados combinadores de ópticas y dificultad para aplicar optimizaciones) quedan resueltas con esta aproximación, tal y como se muestra en la discusión. En la siguiente y última parte de este documento se concluye el trabajo de investigación y se apunta hacia trabajo futuro.

Capítulo 6

El lenguaje Optica

Este capítulo desarrolla el **Obj. 3** (sección 1.4), con el que se pretende diseñar un lenguaje de consultas genéricas, inspirado en las abstracciones y los combinadores existentes en las librerías industriales de ópticas, postulándose como una alternativa a las técnicas convencionales de LINQ. Concretamente, las contribuciones que se recogen a lo largo de este capítulo son las siguientes:

- Se propone una alternativa más homogénea para la composición de ópticas (sección 6.1) que evita mezclar el estilo tan característico de composición de las ópticas con el de otras librerías como la estándar de colecciones en Scala.
- Se presenta Optica, un DSL que proporciona una especificación formal para un subconjunto de ópticas: getters, affine folds y folds. La sintaxis y el sistema de tipos del lenguaje formalizan los combinadores de composición y consultas de una forma abstracta (sección 6.2.1). Sus semánticas denotacionales son dadas en términos de ópticas concretas (sección 6.2.3). Se muestra como implementar consultas genéricas sobre modelos de ópticas abstractos (sección 6.2.2).
- La especificación abstracta de las ópticas de sólo lectura de Optica posibilita la definición de representaciones alternativas y no estándar. Se proporcionan tres interpretaciones que pretenden ilustrar las capacidades de Optica como un lenguaje general de consultas para hacer LINQ:
    - Una interpretación a XQuery [128], que permite la traducción de consultas de Optica en expresiones de este lenguaje (sección 6.3). Se muestra la idoneidad de Optica para lidiar con fuentes de datos anidadas, habituales en bases de datos NoSQL orientadas a documentos.
    - Una interpretación a SQL, que genera expresiones SQL a partir de consultas de Optica (seccion 6.4). Esta semántica no estándar se apoya en el concepto de *tripleta*, donde se normalizan las expresiones de Optica para facilitar su traducción final a SQL. Las semánticas propuestas trabajan de forma similar a la aproximación de normalización por evaluación mostrada en SQUR [73]. La principal diferencia reside

en que SQUR se corresponde con un cálculo relacional mientras que este trabajo se centra en ópticas, más cercanas al algebra relacional.

- Una interpretación a T-LINQ [19] que genera consultas basadas en comprehensions (ver sección 2.5). Esta semántica no estándar tiene como objetivo mostrar cómo usar Optica como un lenguaje de alto nivel para modelos anidados junto con lenguajes basados en comprehensions (sección 6.5).

6.1. Composición homogénea de ópticas

La sección 2.1.4 mostraba una implementación de `expertiseFl` que mezclaba la librería de ópticas introducida en la sección 2.1.1 con la librería estándar de colecciones de Scala, lo que derivaba en una implementación un tanto confusa por la mezcla de aproximaciones. La observación 13 pretendía homogeneizar la implementación mediante la adopción de ópticas para definir el predicado. El código resultante se vuelve a mostrar a continuación:

```
val expertiseFl: Fold[Org, String] =
  departments ⋙ filtered(
    employees.all((tasks ⋙ tsk).elem("abstract").get).get) ⋙ dpt
```

Como se puede apreciar, y a pesar de la homogeneización alcanzada, es necesario realizar varias invocaciones a `get` para poder transformar las diversas ópticas que forman el predicado en consultas (o funciones), que es lo que los métodos `all` y `filtered` esperan recibir como argumento. Esta forma de componer ópticas contrasta con el contenido de la observación 14, donde se manifiesta la importancia de separar las definiciones de las ópticas de las consultas que éstas denotan.

En esta sección se pretende dar un paso más en la homogeneización mediante la introducción de un cambio muy sutil: cambiar los predicados por getters que seleccionen booleanos. La implementación de `filtered` (figura 2.5) tendría que ser adaptada de la siguiente manera:

```
def filtered[S](p: Getter[S, Boolean]): AffineFold[S, S] =
  AffineFold(s => if (p.get(s)) Some(s) else None)
```

El resto de combinadores que se ven afectados por el cambio se pueden encontrar en la figura 6.1. En general, la adaptación de los diversos métodos resulta evidente, pero la implementación de `elem` requiere una explicación más detallada. Al no poder producir una función como predicado, resulta necesario utilizar los combinadores existentes para proporcionar la implementación análoga. La siguiente derivación muestra la adaptación, en la que se parte de un predicado en el que existen abstracciones lambda explícitas y se llega a una versión final en la que quedan fuera de la expresión:

```
  fl.any(Getter(s => s == a))
≃ [definition of `id` getter]
  fl.any(Getter(s => id.get(s) == a)
≃ [definition of `like` getter]
  fl.any(Getter(s => id.get(s) == like(a).get(s))
≃ [definition of `equal` getter]
  fl.any(id === like(a))
```

Con esto se demuestra que ambas expresiones son equivalentes, pero la última de ellas reutiliza los combinadores existentes. Con esto se llega a un estilo más

homogéneo en el que la totalidad de la expresión se compone a partir de ópticas, evitando recurrir a librerías externas como la de colecciones, al menos en la programación de la lógica. Los nuevos combinadores nos permiten implementar `expertiseFl` de la siguiente manera:

```
val expertiseFl: Fold[Org, String] =
  departments ⋙ filtered(
    employees.all((tasks ⋙ tsk).elem("abstract"))) ⋙ dpt
```

Merece la pena destacar la ausencia de las invocaciones a `get` en esta definición. Este estilo de programación con ópticas inspira la implementación del lenguaje Optica, como se mostrará a lo largo de las próximas secciones.

```
def all[S, A](fl: Fold[S, A])(p: Getter[A, Boolean]): Getter[S, Boolean] =
  (fl ⋙ filtered(p.not)).empty

def any[S, A](fl: Fold[S, A])(p: Getter[A, Boolean]): Getter[S, Boolean] =
  fl.all(p.not).not

def elem[S, A: Equal](fl: Fold[S, A])(a: A): Getter[S, Boolean] =
  fl.any(id === like(a))
```

Figura 6.1: Utilización de getters como predicados, en lugar de funciones.

6.2. Un lenguaje inspirado en ópticas concretas

Optica es un lenguaje específico de dominio que pretende llevar las ópticas y sus patrones de diseño a un nivel de abstracción más alto. En particular, su principal objetivo es el de permitir la definición de consultas genéricas, que más tarde puedan ser traducidas en consultas sobre otras infraestructuras, como XQuery (consultas sobre documentos XML) o SQL (consultas sobre tablas relacionales). Para ello, los getters, affine folds y folds que componen el nuevo lenguaje dejan de ser funciones que trabajan sobre estructuras de datos inmutables y pasan a ser dominios semánticos abstractos. Más tarde serán concretados para las diferentes infraestructuras a las que se pretende dar soporte.

Esta sección introduce los aspectos más fundamentales del lenguaje. En primer lugar, se mostrará la sintaxis y el sistema de tipos de Optica, donde se declaran las primitivas y combinadores estándar. En segundo lugar, se introducirán las extensiones necesarias para dar cabida a los ejemplos particulares, imprescindibles para poder definir consultas genéricas sobre un dominio. Después, proporcionaremos la semántica estándar, que nos permitirá recuperar el comportamiento habitual de las ópticas, es decir, el que nos permite lidiar con estructuras de datos inmutables. Todos estos aspectos serán introducidos usando una notación conceptual, por lo que el último paso que tomaremos en esta sección será el de implementar estas ideas en Scala.

6.2.1. Sintaxis y sistema de tipos

La sintaxis de Optica queda recogida en la figura 6.2. La parte superior nos muestra los tipos base (enteros, booleanos y cadenas de texto), los tipos de ópticas soportados (getter, affine fold y fold) y los tipos asociados a las consultas

Base types	$b ::= \mathbb{N} \mid \mathbb{B} \mid \mathbb{S}$
Model types	$t ::= b \mid (t, t)$
Optic Types	$s ::= \mathit{getter}\ t\ t \mid \mathit{affine}\ t\ t \mid \mathit{fold}\ t\ t$
Query Types	$u ::= t \to t \mid t\ \to \mathit{option}\ t \mid t \to \mathit{list}\ t$
Constants	c (of base type)
Optic Expressions	$e ::= \mathit{id}_{gt} \mid \mathit{id}_{af} \mid \mathit{id}_{fl}$
	$\mid e \ggg_{gt} e \mid e \ggg_{af} e \mid e \ggg_{fl} e$
	$\mid \mathit{like}\ c \mid \mathit{not}\ e \mid e > e \mid e == e \mid e - e \mid e\ \ast\!\ast\!\ast\ e$
	$\mid \mathit{filtered}\ e \mid \mathit{nonEmpty}\ e$
	$\mid \mathit{to}_{af}\ e \mid \mathit{to}_{fl}\ e$
Query Expressions	$q ::= \mathit{get}\ e \mid \mathit{preview}\ e \mid \mathit{getAll}\ e$

Figura 6.2: Sintaxis de Optica

(funciones de selección). La parte inferior contiene el repertorio de términos que compone el lenguaje, que tal y como puede apreciarse, están definidos en sintonía con los combinadores concretos. No obstante, existen ciertos matices diferenciadores que merece la pena destacar:

- Una constante (c) por sí misma no es una expresión válida en Optica. Más adelante se verá cómo explotar *like* para representar constantes en el lenguaje, enmascarándolas con forma de getters. Esta práctica mejora la composicionalidad del lenguaje ya que posibilita que las constantes puedan interoperar con otras ópticas reutilizando los combinadores ya existentes.
- La sintaxis no utiliza la notación de orientación a objetos idiomática en Scala. En su lugar, se adopta una notación prefija, que es más habitual en este tipo de definiciones matemáticas, tal y como muestra la literatura.
- Los métodos *all*, *any*, *elem* y *empty* no se incluyen como primitivas del lenguaje. En su lugar, se introducen como definiciones derivadas, como puede apreciarse en la figura 6.3.
- La versión actual de Optica sólo permite expresiones de consultas atómicas, es decir, a diferencia de las expresiones de ópticas, no es posible componerlas. En el capítulo 9 mostraremos que este aspecto podría estar muy ligado a la investigación realizada en el capítulo 3.

El sistema de tipos queda reflejado en la figura 6.4, donde α, β y γ representan tipos del modelo (ver figura 6.2). A diferencia de T-LINQ [16] o QUEΛ [73], Optica no introduce términos para variables. Por tanto, sus reglas de tipado se simplifican ligeramente, ya que omiten el entorno de variables '$\Gamma \vdash$' tan característico en este tipo de formalizaciones. Como se puede apreciar, las reglas se estructuran en cuatro grupos, que se corresponden con los getters, los affine folds, los folds y finalmente las consultas que derivan de estos. Consideramos interesantes los casos de id_* y *like*, por tratarse de términos que forman expresiones de ópticas a partir de la nada. El resto de combinadores deberían ser

def *empty fl* =
  nonEmpty fl $\ggg$ *not*

def *all fl p* =
  empty (*fl* $\ggg$ *filtered* (*not p*))

def *any fl p* =
  not (*all fl* (*not p*))

def *elem fl u* =
  any fl (*id* == *like a*)

Figura 6.3: Definiciones derivadas a partir de las primitivas de Optica

$$\frac{}{id_{gt} : getter\ \alpha\ \alpha}\ id_{gt} \qquad \frac{g_1 :\ getter\ \alpha\ \beta \quad g_2 :\ getter\ \beta\ \gamma}{g_1\ \ggg_{gt}\ g_2 :\ getter\ \alpha\ \gamma}\ \ggg_{gt}$$

$$\frac{g_1 :\ getter\ \alpha\ \beta \quad g_2 :\ getter\ \alpha\ \gamma}{g_1\ ***\ g_2 :\ getter\ \alpha\ (\beta,\ \gamma)}\ *** \qquad \frac{b :\ \beta \quad \beta \in \text{base types}}{like\ b :\ getter\ \alpha\ \beta}\ like$$

$$\frac{g :\ getter\ \alpha\ \mathbb{B}}{not\ g :\ getter\ \alpha\ \mathbb{B}}\ not \qquad \frac{g_1 :\ getter\ \alpha\ \mathbb{N} \quad g_2 :\ getter\ \alpha\ \mathbb{N}}{g_1 > g_2 :\ getter\ \alpha\ \mathbb{B}}\ >$$

$$\frac{g_1 :\ getter\ \alpha\ \beta \quad g_2 :\ getter\ \alpha\ \beta}{g_1 == g_2 :\ getter\ \alpha\ \mathbb{B}}\ == \qquad \frac{g_1 :\ getter\ \alpha\ \mathbb{N} \quad g_2 :\ getter\ \alpha\ \mathbb{N}}{g_1 - g_2 :\ getter\ \alpha\ \mathbb{N}}\ -$$

$$\frac{}{id_{af} : affine\ \alpha\ \alpha}\ id_{af} \qquad \frac{a_1 :\ affine\ \alpha\ \beta \quad a_2 :\ affine\ \beta\ \gamma}{a_1\ \ggg_{af}\ a_2 :\ affine\ \alpha\ \gamma}\ \ggg_{af}$$

$$\frac{p :\ getter\ \alpha\ \mathbb{B}}{filtered\ p :\ affine\ \alpha\ \alpha}\ filtered \qquad \frac{g :\ getter\ \alpha\ \beta}{to_{af}\ g : affine\ \alpha\ \beta}\ to_{af}$$

$$\frac{}{id_{fl} : fold\ \alpha\ \alpha}\ id_{fl} \qquad \frac{f_1 :\ fold\ \alpha\ \beta \quad f_2 :\ fold\ \beta\ \gamma}{f_1\ \ggg_{fl}\ f_2 :\ fold\ \alpha\ \gamma}\ \ggg_{fl}$$

$$\frac{f :\ fold\ \alpha\ \beta}{nonEmpty\ f :\ getter\ \alpha\ \mathbb{B}}\ nonEmpty \qquad \frac{a :\ affine\ \alpha\ \beta}{to_{fl}\ a : fold\ \alpha\ \beta}\ to_{fl}$$

$$\frac{g :\ getter\ \alpha\ \beta}{get\ g :\ \alpha \to \beta}\ get \qquad \frac{a :\ affine\ \alpha\ \beta}{preview\ a :\ \alpha \to option\ \beta}\ preview$$

$$\frac{f :\ fold\ \alpha\ \beta}{getAll\ f :\ \alpha \to list\ \beta}\ getAll$$

Figura 6.4: Sistema de tipos de Optica

triviales, debido a su correspondencia con los combinadores que fueron introducidos en la sección 2.1.1[1]. En cuanto a los términos que producen consultas, es importante destacar que no surgen de los companion objects, sino de las propias definiciones de las ópticas concretas como case classes. La formalización de estos términos hace necesario introducir las funciones como un nuevo dominio semántico para Optica, además de los getters, affine folds y folds. Sin embargo, es importante destacar que la parte del lenguaje que produce expresiones de ópticas es puramente de primer orden, es decir, no se requieren expresiones lambda para construir expresiones de ópticas.

6.2.2. Extensiones del lenguaje y consultas genéricas

En la sección 2.1.4 se pudo ver cómo las consultas generadas mediante ópticas concretas no sólo hacen referencia a combinadores estándar, sino también a ópticas específicas que forman parte de los modelos asociados a los ejemplos. Si pretendemos componer queries genéricas, tenemos que encontrar la manera de acomodar este aspecto. Para esta tarea, se propone que cada ejemplo extienda el lenguaje con nuevos tipos y términos que recojan las necesidades del dominio introducido. La figura 6.5 muestra la extensión que se requiere para adaptar las ópticas asociadas al ejemplo de las parejas. Primero, se introducen las nuevas entidades recogidas en el dominio (*Couples*, *Person*, etc.). Después se muestran los términos que representan las ópticas específicas en el dominio (*couples*, *her*, etc.). Finalmente, se incluyen las reglas de tipos que indican los tipos denotados por dichos términos, es decir, la variante de óptica y los tipos asociados al *todo* y a la *parte*.

$$\begin{array}{lll} \text{Entity Types} & t ::= & \mathit{Couples} \mid \mathit{Couple} \mid \mathit{Person} \\ \text{Optic Expressions} & e ::= & \mathit{couples} \mid \mathit{her} \mid \mathit{him} \mid \mathit{name} \mid \mathit{age} \end{array}$$

$$\frac{}{\mathit{couples} : \mathit{fold}\ \mathit{Couples}\ \mathit{Couple}}\ \mathit{couples} \qquad \frac{}{\mathit{her} : \mathit{getter}\ \mathit{Couple}\ \mathit{Person}}\ \mathit{her}$$

$$\frac{}{\mathit{him} : \mathit{getter}\ \mathit{Couple}\ \mathit{Person}}\ \mathit{him} \qquad \frac{}{\mathit{name} : \mathit{getter}\ \mathit{Person}\ \mathbb{S}}\ \mathit{name}$$

$$\frac{}{\mathit{age} : \mathit{getter}\ \mathit{Person}\ \mathbb{N}}\ \mathit{age}$$

Figura 6.5: Extensión de sintaxis y sistema de tipos para el ejemplo *couples*

Una vez que nuestro lenguaje recoge tanto los combinadores definidos en el *core* como las extensiones que permiten modelar los ejemplos particulares, la composición de consultas genéricas ya debería ser viable.

Definición 31. Las versiones genéricas para *differencesFl* (expresión que denota una óptica) y *differences* (expresión que denota una consulta) quedan im-

[1] No es necesario introducir reglas para los combinadores *any*, *all*, *elem* y *empty*, ya que se derivan a partir de otros términos primitivos.

Entity Types	$t +:= \mathit{Org} \mid \mathit{Department} \mid \mathit{Employee} \mid \mathit{Task}$
Optic Expressions	$e +:= \mathit{departments} \mid \mathit{dpt} \mid \mathit{employees} \mid \mathit{emp} \mid \mathit{tasks} \mid \mathit{tsk}$

$$\frac{}{\mathit{departments} : \mathit{fold}\ \mathit{Org}\ \mathit{Department}}\ \mathit{departments} \qquad \frac{}{\mathit{dpt} : \mathit{getter}\ \mathit{Department}\ \mathbb{S}}\ \mathit{dpt}$$

$$\frac{}{\mathit{employee} : \mathit{fold}\ \mathit{Department}\ \mathit{Employee}}\ \mathit{employees} \qquad \frac{}{\mathit{emp} : \mathit{getter}\ \mathit{Employee}\ \mathbb{S}}\ \mathit{emp}$$

$$\frac{}{\mathit{tasks} : \mathit{getter}\ \mathit{Employee}\ \mathit{Task}}\ \mathit{tasks} \qquad \frac{}{\mathit{tsk} : \mathit{getter}\ \mathit{Task}\ \mathbb{S}}\ \mathit{tsk}$$

Figura 6.6: Extensión de sintaxis y sistema de tipos para ejemplo *org*

plementadas de la siguiente manera:

def *differencesFl* =
$\mathit{couples} \ggg \mathit{to_{fl}}\ (\mathit{filtered}\ ((\mathit{her} \ggg \mathit{age}) > (\mathit{snd} \ggg \mathit{age})) \ggg$
$\mathit{to_{af}}\ ((\mathit{her} \ggg \mathit{name}) *** ((\mathit{her} \ggg \mathit{age}) - (\mathit{snd} \ggg \mathit{age}))))$

def *differences* =
getAll differencesFl

Como se puede observar, dichas implementaciones son básicamente las mismas que las que se mostraron en la sección 2.1.4, siempre y cuando se tengan en cuenta los puntos enumerados en la sección 6.2.1. Se han omitido las invocaciones a $\mathit{to_{af}}$ y $\mathit{to_{fl}}$, donde asumiremos que su aplicación se lleva a cabo de forma implícita.

Se puede llevar a cabo el mismo experimento para el ejemplo de la organización siguiendo el mismo proceso. La extensión del lenguaje se recoge en la figura 6.6, donde de nuevo aparecen las entidades propias del dominio, sus términos y sus reglas de tipado. Bajo esta extensión, somos capaces de definir la correspondiente consulta genérica.

Definición 32. Las versiones genéricas para *expertiseFl* (expresión que denota una óptica) y *expertise* (expresión que denota una consulta) quedan implementadas de la siguiente manera:

def *expertiseFl* =
$\mathit{departments} \ggg \mathit{to_{fl}}\ (\mathit{filtered}\ (\mathit{all}\ \mathit{employees}\ (\mathit{elem}\ (\mathit{tasks} \ggg \mathit{to_{fl}}\ (\mathit{to_{af}}\ \mathit{tsk}))\ \text{`}\mathit{abstract}\text{'})) \ggg \mathit{to_{af}}\ \mathit{dpt})$

def *expertise* =
getAll expertiseFl

Las definiciones previas han introducido consultas genéricas que no están comprometidas con ningún dominio semántico. En el resto del capítulo mostraremos cómo reutilizarlas, traduciéndolas en consultas específicas para diversas infraestructuras, como expresiones XQuery (consultas sobre documentos XML) o consultas SQL (consulta sobre tablas relacionales). Antes de esto, recuperaremos el comportamiento habitual de las ópticas, que nos permitirá producir consultas sobre estructuras de datos inmutables en memoria, lo que se corresponde con la semántica estándar de Optica.

$$
\begin{array}{ll}
\mathcal{T}[\mathbb{N}] & = \texttt{Int} \\
\mathcal{T}[\mathbb{B}] & = \texttt{Boolean} \\
\mathcal{T}[\mathbb{S}] & = \texttt{String} \\
\mathcal{T}[(\alpha, \beta)] & = \texttt{(}\mathcal{T}[\alpha],\ \mathcal{T}[\beta]\texttt{)} \\
\mathcal{T}[\textit{getter}\ \alpha\ \beta] & = \texttt{Getter}[\mathcal{T}[\alpha],\ \mathcal{T}[\beta]] \\
\mathcal{T}[\textit{affine}\ \alpha\ \beta] & = \texttt{Affine}[\mathcal{T}[\alpha],\ \mathcal{T}[\beta]] \\
\mathcal{T}[\textit{fold}\ \alpha\ \beta] & = \texttt{Fold}[\mathcal{T}[\alpha],\ \mathcal{T}[\beta]] \\
\mathcal{T}[\alpha \rightarrow\ \beta] & = \mathcal{T}[\alpha] \Rightarrow\ \mathcal{T}[\beta] \\
\mathcal{T}[\alpha \rightarrow \textit{option}\ \beta] & = \mathcal{T}[\alpha] \Rightarrow\ \texttt{Option}[\mathcal{T}[\beta]] \\
\mathcal{T}[\alpha \rightarrow \textit{list}\ \beta] & = \mathcal{T}[\alpha] \Rightarrow\ \texttt{List}[\mathcal{T}[\beta]]
\end{array}
$$

Figura 6.7: Dominios semánticos estándar de Optica

6.2.3. Semántica estándar

Cuando se define un nuevo lenguaje, es habitual comenzar definiendo su sintaxis y sistema de tipos, y más tarde se define su semántica. En nuestro caso particular, el proceso se ha invertido: partíamos de unas semánticas deseadas (ópticas y acciones) y hemos creado una sintaxis abstracta y un sistema de tipos que imita su estructura. Por tanto, el único trabajo a realizar en esta sección es el de formalizar la conexión existente entre la sintaxis y el sistema de tipos de Optica y las ópticas concretas —su semántica de destino. Para esta tarea, se proporcionan funciones semánticas $\mathcal{T}$ (figura 6.7) y $\mathcal{E}$ (figura 6.8). La primera de ellas se destina a traducir cada tipo de Óptica en su correspondiente dominio semántico. Como se puede apreciar, $\mathcal{T}$ simplemente adapta los tipos de Optica en sus análogos en Scala[2]. Por su parte, la segunda función traduce expresiones que denotan un tipo t en elementos del dominio semántico $\mathcal{T}(t)$. Dado este escenario, la implementación de $\mathcal{E}$ resulta muy sencilla, donde simplemente se adaptan los combinadores de Optica en los combinadores que se introdujeron en la sección 2.1.4. Merece la pena puntualizar que $\oplus$ no es más que un símbolo con el que pretendemos unificar los diversos combinadores binarios (`>`, `-`, etc.).

Al igual que hemos proporcionado una evaluación para los términos que conforman el *core* del lenguaje, también requerimos la evaluación de los términos introducidos por las extensiones asociadas a los diferentes ejemplos. Por ejemplo, la figura 6.9 nos muestra cómo la evaluación de estos términos consiste en su traducción a las ópticas específicas del ejemplo de las parejas, definidas en `CoupleModel` (sección 2.1.4). La evaluación de la extensión asociada al ejemplo de la organización es muy similar y no se muestra por simplicidad.

Una vez descrita la semántica estándar de Optica, deberíamos de ser capaces de utilizar $\mathcal{T}$ y $\mathcal{E}$ para proceder con la traducción de consultas genéricas en funciones planas. Mostramos la evaluación de *differences* (Definición 31) justo

[2]Scala no ofrece soporte nativo para números naturales. Por mantener la implementación sencilla, se utilizará el tipo `Int` para cubrir este aspecto.

$$
\begin{array}{lcl}
\mathcal{E}[id_{gt} : \ getter\ \alpha\ \alpha] & = & \texttt{Getter.id} \\
\mathcal{E}[g \ \ggg_{gt} h : getter\ \alpha\ \gamma] & = & \texttt{Getter.andThen}(\mathcal{E}[g : getter\ \alpha\ \beta],\ \mathcal{E}[h : getter\ \beta\ \gamma]) \\
\mathcal{E}[g \ *** \ h : \ getter\ \alpha\ (\beta,\ \gamma)] & = & \texttt{Getter.fork}(\mathcal{E}[g : getter\ \alpha\ \beta],\ \mathcal{E}[h : getter\ \alpha\ \gamma]) \\
\mathcal{E}[like\ b : \ getter\ \alpha\ \beta] & = & \texttt{Getter.like}(b) \\
\mathcal{E}[not\ g : \ getter\ \alpha\ \mathbb{B}] & = & \texttt{Getter.not}(\mathcal{E}[g : getter\ \alpha\ \mathbb{B}]) \\
\mathcal{E}[g \oplus h : \ getter\ \alpha\ \delta] & = & \texttt{Getter.} \oplus (\mathcal{E}[g : getter\ \alpha\ \beta],\ \mathcal{E}[h : getter\ \alpha\ \gamma]) \\
 & & \\
\mathcal{E}[id_{af} : \ affine\ \alpha\ \alpha] & = & \texttt{AffineFold.id} \\
\mathcal{E}[g \ \ggg_{af} h : affine\ \alpha\ \gamma] & = & \texttt{AffineFold.andThen}(\mathcal{E}[g : affine\ \alpha\ \beta],\ \mathcal{E}[h : affine\ \beta\ \gamma]) \\
\mathcal{E}[filtered\ p : \ affine\ \alpha\ \alpha] & = & \texttt{AffineFold.filtered}(\mathcal{E}[p : getter\ \alpha\ \mathbb{B}]) \\
\mathcal{E}[to_{af}\ g : \ affine\ \alpha\ \beta] & = & \texttt{AffineFold.to}_{af}(\mathcal{E}[g : getter\ \alpha\ \beta]) \\
 & & \\
\mathcal{E}[id_{fl} : \ fold\ \alpha\ \alpha] & = & \texttt{Fold.id} \\
\mathcal{E}[g \ \ggg_{fl} h : fold\ \alpha\ \gamma] & = & \texttt{Fold.andThen}(\mathcal{E}[g : fold\ \alpha\ \beta],\ \mathcal{E}[h : fold\ \beta\ \gamma]) \\
\mathcal{E}[nonEmpty\ g : \ getter\ \alpha\ \mathbb{B}] & = & \texttt{Fold.nonEmpty}(\mathcal{E}[g : \ fold\ \alpha\ \beta]) \\
\mathcal{E}[to_{fl}\ a : \ fold\ \alpha\ \beta] & = & \texttt{Fold.to}_{fl}(\mathcal{E}[g : affine\ \alpha\ \beta]) \\
 & & \\
\mathcal{E}[get\ g : \ \alpha \to \beta] & = & \mathcal{E}[g : getter\ \alpha\ \beta]\texttt{.get} \\
\mathcal{E}[preview\ g : \ \alpha \to option\ \beta] & = & \mathcal{E}[g : affine\ \alpha\ \beta]\texttt{.preview} \\
\mathcal{E}[getAll\ g : \ \alpha \to list\ \beta] & = & \mathcal{E}[g : fold\ \alpha\ \beta]\texttt{.getAll}
\end{array}
$$

Figura 6.8: Semántica estándar de Optica

$$
\begin{array}{lcl}
\mathcal{T}[Couples] & = & \texttt{Couples} \\
\mathcal{T}[Couple] & = & \texttt{Couple} \\
\mathcal{T}[Person] & = & \texttt{Person} \\
 & & \\
\mathcal{E}[couples : \ fold\ Couples\ Couple] & = & \texttt{CoupleModel.couples} \\
\mathcal{E}[her : getter\ Couple\ Person] & = & \texttt{CoupleModel.her} \\
\mathcal{E}[him : getter\ Couple\ Person] & = & \texttt{CoupleModel.him} \\
\mathcal{E}[name : getter\ Person\ \mathbb{S}] & = & \texttt{CoupleModel.name} \\
\mathcal{E}[age : getter\ Person\ \mathbb{N}] & = & \texttt{CoupleModel.age}
\end{array}
$$

Figura 6.9: Semántica estándar de la extensión asociada al ejemplo *couples*

a continuación:

def *differencesR* : `Couples` ⇒ `List[(String, Int)]` =
  $\mathcal{E}$[*differences* : *Couples* → *list* (𝕊, ℕ)]

El valor resultante es una función de Scala que trabaja con estructuras de datos inmutables, tal y como ocurre en la adaptación de *expertise* (Definición 32):

def *expertiseR u* : `Org` ⇒ `List[String]` =
  $\mathcal{E}$[*expertise u* : *Org* → *list* 𝕊]

Las funciones producidas bajo esta evaluación son exactamente las mismas que sus análogas de la sección 2.1.4.

6.3. XQuery

Hasta ahora hemos visto que las ópticas nos permiten manipular estructuras de datos inmutables de forma modular y elegante, y que sus combinadores se pueden empaquetar en un DSL, al que hemos bautizado como Optica. La semántica estándar asociada a dicho lenguaje está definida en términos de ópticas concretas. Sin embargo, esto no supone un gran hito desde la perspectiva de LINQ, ya que el estado de las aplicaciones reales es principalmente mantenido mediante bases de datos, servicios web, etc. Durante ésta y las próximas secciones mostraremos como reutilizar expresiones de Optica para generar consultas específicas sobre fuentes de datos externas. En particular, esta sección nos mostrará que es posible dotar a los getters, affine folds y folds de una semántica no estándar en términos de expresiones XQuery. Antes de mostrar esta semántica, nos parece conveniente introducir brevemente el entorno de XML/XQuery [128]. Para guiar las explicaciones proporcionaremos los modelos XML asociados a los ejemplos de las parejas y de la organización e implementaremos en XQuery consultas análogas para `differences` y `expertise` en un estilo idiomático. Nos basaremos en estas definiciones como punto de partida para implementar la semántica no estándar, donde tendremos que contemplar ciertas asunciones que iremos describiendo según vayan surgiendo.

6.3.1. Antecedentes

La adaptación de objetos en elementos XML no es una tarea sencilla, tal y como revela [78]. En la figura 6.10 se puede ver una posible forma de codificar el estado del ejemplo de las parejas en un documento XML. Concretamente, el documento contiene un elemento raíz `<xml/>` del que cuelgan las parejas, y a su vez de éstas cuelgan las mujeres `<her/>` y los hombres `<him/>` que las forman. En la parte más profunda del documento nos encontramos con `<name/>` y `<age/>`, que son elementos sencillos en los que se alojan valores primitivos.

Es habitual que un documento XML venga acompañado por un esquema XSD cuando queremos validar la información dispuesta en él. Podemos encontrar el esquema asociado al documento de las parejas en la figura 6.11. Entre otros aspectos, se encarga de evitar que se definan personas sin un elemento `<name/>`, proporcionar valores no numéricos para el elemento `<age/>` y definir varias ocurrencias del elemento `<her/>` como parte de una pareja `<couple/>`. Más

```
<xml>
    <couple>
        <her><name>Alex</name><age>60</age></her>
        <him><name>Bert</name><age>55</age></him>
    </couple>
    <couple>
        <her><name>Cora</name><age>33</age></her>
        <him><name>Drew</name><age>31</age></him>
    </couple>
    <couple>
        <her><name>Edna</name><age>21</age></her>
        <him><name>Fred</name><age>60</age></him>
    </couple
</xml>
```

Figura 6.10: Parejas representadas como un documento XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="xml">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="couple" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="her">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="name" type="xs:string"/>
                                        <xs:element name="age" type="xs:positiveInteger"/>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="him">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="name" type="xs:string"/>
                                        <xs:element name="age" type="xs:positiveInteger"/>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figura 6.11: Esquema XSD para validar documentos XML de parejas.

adelante, se mostrará la relevancia de este tipo de esquemas para componer consultas XQuery.

Una vez que ya se ha adaptado el ejemplo de las parejas en XML, nos gustaría poder lanzar consultas sobre él, análogas a `differences`, para lo que utilizaremos el lenguaje de consultas XQuery. La consulta debería ser capaz de recopilar el nombre y la diferencia de edad de todas aquellas parejas donde la mujer sea mayor que el hombre. Como ya hemos visto con anterioridad, no es una consulta que produzca un valor único, como un número o un booleano, sino más bien debería producir una secuencia de nodos, es decir, la salida generada debería presentarse como un documento XML. El siguiente fragmento XML podría representar la salida que estamos buscando:

```
<xml>
    <tuple>
        <fst><name>Alex</name></fst>
        <snd>5</snd>
    </tuple>
    <tuple>
        <fst><name>Cora</name></fst>
        <snd>2</snd>
    </tuple>
</xml>
```

Para poder unificar el nombre y la edad se ha utilizado un elemento `<tuple/>`, que contiene subelementos `<fst/>` y `<snd/>` que hacen las veces de proyecciones, donde se guardan los valores finales. Se podría utilizar la siguiente expresión XQuery para recuperar dicha salida[3]:

```
/xml/couple[her/age > him/age]/<tuple>
                                 <fst>{her/name}</fst>
                                 <snd>{her/age - him/age}</snd>
                               </tuple>
```

Describimos los elementos principales de esta consulta en la siguiente lista:

- Una de las consultas más relevantes en XQuery es `/`, que nos brinda acceso al denominado nodo *document*, que se puede ver como el punto de entrada al documento. Teniendo en cuenta que los documentos XML son esencialmente estructuras de datos anidadas, XQuery proporciona una sintaxis muy directa para acceder a elementos anidadas. Por ejemplo `/xml/couple` selecciona todos los elementos `<couple/>` que estan colgando de un elemento `<xml/>`, que a su vez debe ser accesible desde el nodo documento.

- Las expresiones XQuery pueden contener filtros para enriquecer las consultas, cuyos predicados se disponen entre corchetes. Por ejemplo, `[her/age > him/age]` es un filtro que se aplica sobre `/xml/couple` para descartar todas aquellas parejas donde la edad de la mujer no supera a la del hombre. Merece la pena destacar que el operador `>` extrae el valor interno de los elementos para llevar a cabo la comparación. El esquema XSD nos asegura que dichos elementos contienen valores numéricos, por lo que la consulta no debería producir situaciones excepcionales derivados de este aspecto.

- XQuery soporta interpolación de XML para poder estructurar los resultados. En nuestro caso particular se utiliza esta técnica para poner los pares de resultados juntos. De todas las utilidades de XQuery de las que

[3] La consulta se ha dividido en varias líneas para facilitar su lectura, pero algunos intérpretes requieren que ésta consulta aparezca en una única línea.

```
<xml>
    <department>
        <dpt>Product</dpt>
        <employee><emp>Alex</emp><task><tsk>build</tsk></task></employee>
        <employee><emp>Bert</emp><task><tsk>build</tsk></task></employee>
    </department>
    <department>
        <dpt>Quality</dpt>
    </department>
    <department>
        <dpt>Research</dpt>
        <employee>
            <emp>Cora</emp>
            <task><tsk>abstract</tsk></task><task><tsk>build</tsk></task><task><tsk>design</tsk></task>
        </employee>
        <employee>
            <emp>Drew</emp>
            <task><tsk>abstract</tsk></task><task><tsk>design</tsk></task>
        </employee>
        <employee>
            <emp>Edna</emp>
            <task><tsk>abstract</tsk></task><task><tsk>call</tsk></task><task><tsk>design</tsk></task>
        </employee>
    </department>
    <department>
        <dpt>Sales</dpt>
        <employee>
            <emp>Fred</emp>
            <task><tsk>call</tsk></task>
        </employee>
    </department>
</xml>
```

Figura 6.12: Organización representada como un documento XML.

haremos uso en este trabajo, la interpolación de XML es la única que no está también disponible en XPath.

También podríamos estar interesados en adpatar el ejemplo de la organización, junto con la consulta análoga a `expertise`. La figura 6.12 muestra una posible forma de codificar el documento XML con el estado correspondiente. Este documento es válido de acuerdo con el esquema XSD que puede encontrarse en la figura 6.13. La adaptación de la consulta `expertise` debería contener una secuencia con los nombres de todos aquellos departamentos donde todos los empleados saben abstraer. De nuevo, por tratarse de una secuencia, la salida producida será un documento XML como el siguiente:

```
<xml>
    <dpt>Quality</dpt>
    <dpt>Research</dpt>
</xml>
```

Generar la consulta que produce este resultado no es fácil, ya que no existe ningún método estándar en XQuery para comprobar si todos los elementos que están colgando de cierto contexto cumplen cierto predicado. Afortunadamente, podemos implementar el comportamiento deseado en términos de primitivas más simples, tal y como mostramos en la consulta propuesta:

```
/xml/department[not(employee[not(task[tsk = "abstract"])])]/dpt
```

Es complicado entender el propósito de esta consulta, principalmente por la combinación de filtros y negaciones. La consulta muestra nuevos combinadores de XQuery, que describimos a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="xml">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="department" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="dpt" type="xs:string"/>
                            <xs:element name="employee" minOccurs="0" maxOccurs="unbounded">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="emp" type="xs:string"/>
                                        <xs:element name="task" minOccurs="0" maxOccurs="unbounded"
                                                    type="xs:string"/>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figura 6.13: Esquema XSD para validar documentos XML de organizaciones.

- Existen varias invocaciones a la función `not`. Se corresponde con la función de negación que puede encontrarse en muchos lenguajes de programación, pero añade cierta funcionalidad adicional más allá de negar booleanos. En particular, produce `true` si el argumento se corresponde con una secuencia no vacía de elementos y `false` si el argumento se corresponde con una secuencia vacía.

- La consulta también introduce el operador `=`, que permite comparar si dos elementos son iguales. En nuestra consulta, el primer operador es un elemento y por tanto su valor debe ser extraído para llevar a cabo la comparación. El segundo operando es una cadena de caracteres, por lo que no se requiere ninguna extracción. Hay otros literales soportados por XQuery, como números o booleanos.

Por último, nos gustaría introducir otro elemento que hemos ignorado deliberadamente hasta ahora, por no estar presente en la consulta. Se le conoce como *self axis* y se utiliza para referirse al contexto actual. En XQuery, se le representa con un punto (`.`). Esta noción de *self* es neutral en relación al acceso anidado. Por ejemplo, `./couple/./her/.` es equivalente a `couple/her`. Más tarde veremos que esta noción es importante para implementar la semántica no estándar.

6.3.2. Semántica no estándar

Una vez introducidos los fundamentos del entorno XML/XQuery, retomamos nuestro objetivo de traducir expresiones de Optica en expresiones XQuery. Para esta tarea, nos valdremos de la función semántica $\mathcal{E}^{xml}$ como evaluador. Antes de eso, mostraremos $\mathcal{T}^{xml}$, la función que traduce los tipos de Optica en los dominios semánticos asociados a esta infraestructura. Si tenemos en cuenta que queremos producir expresiones XQuery, parece razonable utilizar *XQuery* como

dominio semántico para los tipos asociados a las consultas. También es necesario adaptar los tipos asociados a las ópticas. Sorprendentemente, se utilizará el mismo dominio semántico para ambas familias de tipos (ver observación 23). Por tanto definimos $\mathcal{T}^{xml}$ de la siguiente manera:

$$\mathcal{T}^{xml}[t] = XQuery$$

Por tanto, toda expresión de Optica, independientemente del tipo que denota, evaluará a una expresión XQuery. El resto de la sección completa las definiciones necesarias para la traducción, donde principalmente nos centraremos en la implementación de $\mathcal{E}^{xml}$.

Observación 23. Como acabamos de ver, tanto los tipos asociados a las ópticas como los tipos asociados a las consultas se evalúan al mismo dominio semántico *XQuery*. Si dejamos de lado las facilidades de interpolación de XML, y tal y como veremos en las próximas secciones, la evaluación que proponemos es esencialmente una traducción a XPath. Este lenguaje permite seleccionar partes de un documento XML, al igual que las ópticas permiten seleccionar partes de una estructura de datos inmutables. En este sentido, es natural entender XQuery como una representación no estándar de una óptica.

Evaluación de extensiones del lenguaje

Antes de abordar la implementación de $\mathcal{E}^{xml}$, es necesario declarar todas las asunciones que hacemos sobre la adaptación de modelos de ópticas en modelos XML, donde básicamente adoptaremos la misma convención que hemos seguido en la sección 6.3.1:

- Toda la información cuelga de un elemento `</xml>`, que se posiciona como raíz del documento XML.
- Cada óptica se corresponde con un elemento XML, donde el tipo de óptica determina la cardinalidad del elemento.
- Las ópticas que seleccionan tipos base se adaptan como elementos XML de tipo simple, es decir, aquellos que albergan un valor primitivo.
- Las ópticas que seleccionan tipos entidad se adaptan como elementos XML de tipo complejo, es decir, aquellos que a su vez anidan otros elementos.

Como se puede apreciar, los esquemas XSD de las figuras 6.11 y 6.13 introducen estas restricciones. Estos puntos resultan ser relevantes a la hora de implementar $\mathcal{E}^{xml}$.

Conmenzaremos proporcionando la implementación del evaluador para los términos que extienden Optica con las primitivas propias del ejemplo de las parejas, que se muestran en la figura 6.14. El segundo punto indicaba que las ópticas tienen una correspondencia con los elementos XML, y por tanto se evalúan como una mera selección de elementos. De hecho, el propio nombre de la óptica resulta ser un buen candidato como nombre de la etiqueta. Sin embargo, necesitamos ajustar el nombre en plural de los folds en su singular, como en *couple*, ya que esta información se proporciona como elementos individuales. No mostramos la evaluación para la extensión introducida por el ejemplo de la organización, ya que no aporta ningún valor adicional.

$$
\begin{array}{lcl}
\mathcal{E}^{xml}[\_] & :: & \mathit{XQuery} \\
 & & \\
\mathcal{E}^{xml}[\mathit{couples} : \mathit{fold\ Couples\ Couple}] & = & \mathit{couple} \\
\mathcal{E}^{xml}[\mathit{her} : \mathit{getter\ Couple\ Person}] & = & \mathit{her} \\
\mathcal{E}^{xml}[\mathit{him} : \mathit{getter\ Couple\ Person}] & = & \mathit{him} \\
\mathcal{E}^{xml}[\mathit{name} : \mathit{getter\ Person}\ \mathbb{S}] & = & \mathit{name} \\
\mathcal{E}^{xml}[\mathit{age} : \mathit{getter\ Person}\ \mathbb{N}] & = & \mathit{age}
\end{array}
$$

Figura 6.14: Semántica XQuery para los términos extendidos en *couples*.

Evaluación de primitivas *core*

La evaluación de los combinadores principales de Optica queda recogida en la figura 6.15. Comenzamos con los combinadores para getters. En primer lugar, el término $\ggg_{gt}$ es evaluado como acceso anidado, donde las evaluaciones recursivas de g y h determinan el contexto y la selección en sí misma, respectivamente. Para el caso de $***$ utilizamos la interpolación de XML, donde la evaluación recursiva de g y h se utiliza para cumplimentar los elementos de proyección `<fst/>` y `<snd/>`. La evaluación de id_{gt} produce una referencia self (.), preservando la neutralidad con respecto a la composición vertical. Pasamos ahora a la evaluación de *like*. Teniendo en cuenta que su evaluación introduce ópticas constantes, cuya selección no depende del todo, simplemente se decide producir un literal, donde _*b*_ representa la adaptación del literal b a un literal de XQuery. Finalmente, se puede apreciar cómo *not* se interpreta como la función `not` y los operadores binarios, unificados por el símbolo $\oplus$ se evalúan como el operador XQuery correspondiente.

Seguimos con el siguiente grupo de combinadores, donde nos encontramos con términos que trabajan con affine folds. La implementación de las primitivas de composición e identidad es exactamente la misma a la que hemos introducido para getters. Esta situación que también se dará después para los folds corrobora que no existe distinción entre dominios semánticos para las diversas ópticas. En este sentido, si entendemos *XQuery* como una representación de un affine fold, es perfectamente natural que también pueda ser usada como una representación de un getter. De hecho, la implementación de to_{af} confirma esta intuición. Este bloque de combinadores también incluye *filtered*. Afortunadamente, ya conocemos la existencia de un mecanismo de filtrado en las consultas XQuery, por lo que simplemente evaluamos este combinador en unos corchetes ([]) que contendrán la evaluación recursiva del predicado.

Por último, se presentan los combinadores que trabajan con folds, donde únicamente contemplamos *nonEmpty*. Esta evaluación particular requiere traducir un fold en un getter que selecciona un booleano. Por suerte, XQuery proporciona la función `exists`, que transforma una expresión XQuery en un simple booleano. Esta función ni siquiera fue introducida en la sección 6.3.1, ya que `not` se encargaba de esta tarea por nosotros. En particular, `not(exists(sq))` (donde `sq` denota una secuencia de elementos) es equivalente a `not(sq)`. Sin embargo, durante la evaluación se desconoce si la invocación `exists` denotada por *nonEmpty* será consumida por una función como `not` (denotada por otra expresión), y por tanto necesitamos invocar `exists` explícitamente[4]. El último bloque de la evaluación

[4]Hay diferentes técnicas en la literatura que nos permitirían llevar a cabo este tipo de

$$
\begin{array}{lcl}
\mathcal{E}^{xml}[\_] & :: & \mathit{XQuery} \\
 & & \\
\mathcal{E}^{xml}[id_{gt} : \ \mathit{getter}\ \alpha\ \alpha] & = & . \\
\mathcal{E}^{xml}[g\ \ggg_{gt} h : \mathit{getter}\ \alpha\ \gamma] & = & \mathcal{E}^{xml}[g : \mathit{getter}\ \alpha\ \beta]/\mathcal{E}^{xml}[h : \mathit{getter}\ \beta\ \gamma] \\
\mathcal{E}^{xml}[g\ \ast\!\ast\!\ast\ h : \ \mathit{getter}\ \alpha\ (\beta,\ \gamma)] & = & < \mathit{tuple} > \\
 & & \quad < \mathit{fst} > \mathcal{E}^{xml}[g : \mathit{getter}\ \alpha\ \beta] < /\mathit{fst} > \\
 & & \quad < \mathit{snd} > \mathcal{E}^{xml}[h : \mathit{getter}\ \alpha\ \gamma] < /\mathit{snd} > \\
 & & < /\mathit{tuple} > \\
\mathcal{E}^{xml}[\mathit{like}\ b : \ \mathit{getter}\ \alpha\ \beta] & = & \_b\_ \\
\mathcal{E}^{xml}[\mathit{not}\ g : \ \mathit{getter}\ \alpha\ \mathbb{B}] & = & \mathbf{not}(\mathcal{E}^{xml}[g : \ \mathit{getter}\ \alpha\ \mathbb{B}]) \\
\mathcal{E}^{xml}[g \oplus h : \ \mathit{getter}\ \alpha\ \delta] & = & (\mathcal{E}^{xml}[g : \ \mathit{getter}\ \alpha\ \beta] \oplus \mathcal{E}^{xml}[h : \ \mathit{getter}\ \alpha\ \gamma]) \\
 & & \\
\mathcal{E}^{xml}[id_{af} : \ \mathit{affine}\ \alpha\ \alpha] & = & . \\
\mathcal{E}^{xml}[g\ \ggg_{af} h : \mathit{affine}\ \alpha\ \gamma] & = & \mathcal{E}^{xml}[g : \mathit{affine}\ \alpha\ \beta]/\mathcal{E}^{xml}[h : \mathit{affine}\ \beta\ \gamma] \\
\mathcal{E}^{xml}[\mathit{filtered}\ p : \ \mathit{affine}\ \alpha\ \alpha] & = & .\Big[\mathcal{E}^{xml}[p : \ \mathit{affine}\ \alpha\ \mathbb{B}]\Big] \\
\mathcal{E}^{xml}[to_{af}\ g : \ \mathit{affine}\ \alpha\ \beta] & = & \mathcal{E}^{xml}[g : \mathit{getter}\ \alpha\ \beta] \\
 & & \\
\mathcal{E}^{xml}[id_{fl} : \ \mathit{fold}\ \alpha\ \alpha] & = & . \\
\mathcal{E}^{xml}[g\ \ggg_{fl} h : \mathit{fold}\ \alpha\ \gamma] & = & \mathcal{E}^{xml}[g : \mathit{fold}\ \alpha\ \beta]/\mathcal{E}^{xml}[h : \mathit{fold}\ \beta\ \gamma] \\
\mathcal{E}^{xml}[\mathit{nonEmpty}\ g : \ \mathit{getter}\ \alpha\ \mathbb{B}] & = & \mathbf{exists}(\mathcal{E}^{xml}[g : \ \mathit{fold}\ \alpha\ \beta]) \\
\mathcal{E}^{xml}[to_{fl}\ a : \ \mathit{fold}\ \alpha\ \beta] & = & \mathcal{E}^{xml}[a : \mathit{affine}\ \alpha\ \beta] \\
 & & \\
\mathcal{E}^{xml}[\mathit{get}\ g : \ \alpha \to \beta] & = & /\mathit{xml}/\mathcal{E}^{xml}[g : \ \mathit{getter}\ \alpha\ \beta] \\
\mathcal{E}^{xml}[\mathit{preview}\ g : \ \alpha \to \mathit{option}\ \beta] & = & /\mathit{xml}/\mathcal{E}^{xml}[g : \ \mathit{affine}\ \alpha\ \beta] \\
\mathcal{E}^{xml}[\mathit{getAll}\ g : \ \alpha \to \mathit{list}\ \beta] & = & /\mathit{xml}/\mathcal{E}^{xml}[g : \ \mathit{fold}\ \alpha\ \beta]
\end{array}
$$

Figura 6.15: Semántica no estándar para XQuery

se describe en la próxima sección.

Observación 24. La evaluación de una expresión que denota una óptica, como *differencesFl* (definición 31) o *expertiseFl* (definición 32), deriva en una consulta relativa, es decir, una consulta que no comienza seleccionando el nodo documento (/) y que es relativa al contexto actual. Este tipo de consultas, aún siendo correctas, no generarían ningún resultado si las lanzamos contra los documentos XML que contienen el estado de nuestros ejemplos. Sin embargo, sí que se podrían componer estas consultas con otras que provengan de modelos externos. En la próxima sección se deja esta posibilidad de lado y se introduce el refinamiento final que es necesario para transformar las consultas relativas en consultas absolutas.

Consultas específicas y resultados

La evaluación de las expresiones de Optica que denotan consultas se puede encontrar en el último bloque de combinadores de la figura 6.15. Teniendo en cuenta que $\mathcal{T}^{xml}$ adapta tanto los tipos de ópticas como los de consultas en *XQuery* (observación 23), la evaluación de éstas debería ser bastante directa, ya que únicamente tienen que transformar las consultas relativas resultantes de la evaluación de su argumento en consultas absolutas. Este proceso simplemente requiere añadir `/xml/` al comienzo de las consultas relativas. Recuerda que previamente adoptamos una convención donde los documentos XML debían contener la información asociada colgando de un elemento `<xml/>`. Por tanto, aprovechamos la ocasión para contemplar este aspecto aquí.

Llegado este punto, en el que ya contamos con la implementación completa para $\mathcal{E}^{xml}$, deberíamos ser capaces de producir consultas XQuery específicas a partir de consultas genéricas. La evaluación de *differences* (definición 31) mediante $\mathcal{E}^{xml}[\mathit{differences} : \ \mathit{Couples} \to \mathit{list}\ (\mathbb{S},\ \mathbb{N})]$ produce la expresión XQuery:

```
/xml/couple[her/age > him/age]/<tuple>
                                    <fst>{her/name}</fst>
                                    <snd>{her/age - him/age}</snd>
                                </tuple>
```

Esta expresión es exactamente igual a la que se ha introducido en la sección 6.3.1. La salida generada para la evaluación de *expertise* (definición 32) mediante $\mathcal{E}^{xml}[\mathit{expertise} : \ \mathit{Org} \to \mathbb{S}]$ es la siguiente:

```
/xml/department[not(exists(employee[not(exists(task/tsk[. = "abstract"]))]))]/dpt
```

En esta consulta se puede observar la existencia de una referencia self (.) cuando se compara la tarea con el literal `"abstract"`, consecuencia de la implementación derivada de *elem* que se presentó en la figura 6.3 (donde nos referimos a id_{gt} para recuperar el parámetro del predicado que se pasa como argumento a *any*). Adicionalmente, nos encontramos con llamadas redundantes a la función `exists`. Si se ignoran estos matices, que podrían solucionarse mediante técnicas de normalización [73, 68], la consulta resultante es prácticamente la misma que la que mostramos en la sección 6.3.1, ya que ambas devuelven la misma salida cuando se lanzan contra el mismo documento XML.

---

optimizaciones, pero las ignoramos por simplicidad.

6.4. SQL

SQL es un lenguaje de consultas para fuentes de datos relacionales cuya naturaleza difiere notablemente de la que se asocia con los modelos jerárquicos basados en ópticas o XML. Sin embargo, esto no supone un impedimento para que podamos traducir consultas genéricas de Optica en consultas específicas SQL. Tal y como hicimos en la sección anterior, primero se adaptará el modelo bajo la nueva configuración relacional y se escribirán las consultas SQL de forma manual. Después, nos apoyaremos en las intuiciones obtenidas para presentar las semánticas no estándar, donde el evaluador tendrá que realizar ciertas asunciones para poder generar las correspondientes consultas de manera automática.

6.4.1. Antecedentes

A diferencia de XML, las bases de datos relacionales se organizan en torno a fuentes de datos aplanadas. Como consecuencia, es necesario abordar el *object-relational impedance mismatch* [63] cuando se tratan de adaptar modelos basados en objetos (como el que subyace a las ópticas) en modelos relacionales. Afortunadamente, existen ciertos patrones de diseño que se pueden explotar para llevar a cabo esta adaptación, como los patrones de *Foreign Key Aggregation* o *Foreign Key Association* [69]. Se tomarán como referencia para adaptar, en primer lugar, el ejemplo de las parejas:

```
CREATE TABLE Person (
  name varchar(255) PRIMARY KEY,
  age int NOT NULL
);

CREATE TABLE Couple (
  her varchar(255) NOT NULL,
  him varchar(255) NOT NULL,
  FOREIGN KEY (her) REFERENCES Person(name),
  FOREIGN KEY (him) REFERENCES Person(name)
);
```

Como se puede apreciar, las case classes se convierten en tablas y sus atributos se convierten en columnas. En este nuevo escenario también resulta necesario establecer una distinción entre atributos que contienen tipos base y atributos que contienen entidades. De hecho, los atributos que se refieren a entidades requieren claves para establecer las conexiones correspondientes entre las tablas involucradas, donde se adopta el patrón *Foreign Key Aggregation*. Asumiremos que las figuras 6.16a y 6.16b conforman el estado inicial para estas tablas, donde las columnas que forman la tabla `Couple` están apuntando a nombres existentes en la tabla `Person`.

Ya hemos visto cómo la adaptación de la consulta `differences` en XQuery producía un documento XML como salida (sección 6.3.1). En el escenario relacional de SQL nos encontramos con una situación similar, donde las consultas producen tablas como resultados. De hecho, la figura 6.16c contempla el resultado que se podría esperar de la adaptación de `differences` a una consulta SQL. Proponemos esta consulta como candidata para producir dicho resultado:

```
SELECT w.name, w.age - m.age
FROM Couple c INNER JOIN Person w ON c.her = w.name
              INNER JOIN Person m ON c.him = m.name
WHERE w.age > m.age;
```

name	age
Alex	60
Bert	55
Cora	33
Drew	31
Edna	21
Fred	60

(a) `Person`

her	him
Alex	Bert
Cora	Drew
Edna	Fred

(b) `Couple`

her	diff
Alex	5
Cora	2

(c) `Differences`

Figura 6.16: Datos para las tablas asociadas al ejemplo de las parejas.

Esta declaración está fuertemente diferenciada por tres secciones principales. Primero, describimos **`FROM`**, donde se construye la tabla de la que se nutren las otras dos secciones para recopilar la información requerida. La tabla en cuestión se crea mediante la unión de la tabla `Couple` con dos ocurrencias de la tabla `Person`, que insertan los valores asociados a los miembros de la pareja `her` y `him`. Se introducen variables `c`, `w` y `m` que nos permitirán hacer referencia a estas tablas desde el resto de secciones de la consulta. Pasamos ahora a describir la cláusula **`WHERE`**, donde se introducen filtros que nos permiten descartar todos aquellos registros que no cumplen el criterio: aquellos donde la edad de la mujer no es mayor que la edad del hombre. Por último, la cláusula **`SELECT`** indica las columnas que resultan de interés: el nombre de la mujer y la diferencia de edad.

Ahora se muestra cómo adaptar el ejemplo de la organización. En primer lugar, se crean las tablas asociadas a los departamentos, empleados y tareas:

```
CREATE TABLE Department (
  dpt varchar(255) PRIMARY KEY
);

CREATE TABLE Employee (
  emp varchar(255) PRIMARY KEY,
  dpt varchar(255) NOT NULL,
  FOREIGN KEY (dpt) REFERENCES Department(dpt)
);

CREATE TABLE Task (
  tsk varchar(255) NOT NULL,
  emp varchar(255) NOT NULL,
  FOREIGN KEY (emp) REFERENCES Employee(emp)
);
```

Aunque todos los componentes que aparecen en la definición anterior deberían resultar familiares, existe un cambio importante en la manera en la que se distribuyen las claves foráneas. En el ejemplo de las parejas hemos visto como los getters se mapean en una columna que contiene la clave. Sin embargo, el ejemplo de la organización contiene atributos multivaluados, como `employees` o `tasks`, que no se pueden adaptar como una columna sencilla. En su lugar, se hace uso del patrón *Foreign Key Association*. Asumiremos que el estado inicial para estas tablas es el que se muestra en las figuras 6.17a, 6.17b y 6.17c.

Como ya hemos visto, *Quality* y *Research* son los únicos departamentos donde todos los empleados son capaces de abstraer; por tanto, la adaptación de `expertise` debería producir como resultado el contenido de la figura 6.17d. Proponemos la siguiente consulta SQL para generarlo:

```
SELECT d.dpt
FROM Department AS d
WHERE NOT(EXISTS(SELECT e.*
                 FROM Employee AS e
```

dpt
Product
Quality
Research
Sales

(a) `Department`

emp	dpt
Alex	Product
Bert	Product
Cora	Research
Drew	Research
Edna	Research
Fred	Sales

(b) `Employee`

tsk	emp
build	Alex
build	Bert
abstract	Cora
build	Cora
design	Cora
abstract	Drew
design	Drew
abstract	Edna
call	Edna
design	Edna
call	Fred

(c) `Task`

dpt
Quality
Research

(d) `Expertise`

Figura 6.17: Datos de las tablas asociadas al ejemplo *org*.

```
WHERE NOT(EXISTS(SELECT t.*
                 FROM Task AS t
                 WHERE (t.tsk = "abstract") AND (e.emp = t.emp)))
      AND (d.dpt = e.dpt)));
```

No resulta sencillo entender el contenido de esta consulta, aunque ya tenemos mucho terreno ganado, ya que sigue la misma estructura que la adaptación de `expertise` como una expresión XQuery 6.3.1. De hecho, **`EXISTS`** es una función que devuelve **`True`** siempre y cuando la consulta anidada no produzca un resultado vacío. Si combinamos esta función con **`NOT`** para negar predicados, podremos comprobar si todos los registros satisfacen la condición. Más allá del ruido introducido por este patrón, la existencia de **`EXISTS`** requiere establecer las relaciones precisas que existen entre las variables anidadas y las variables externas, lo cual introduce todavía más complejidad.

6.4.2. Semántica no estándar

El hecho de que SQL tenga ciertas peculiaridades atribuidas es sabido desde hace mucho tiempo. Tal y como plantea [25] en relación con ciertos aspectos de SQL, "there is so much confusion in this area that it is difficult to criticize it coherently". Parte del problema reside en que la formalización de SQL tuvo lugar después de su versión para la industria, y por consiguiente, muchas consideraciones académicas provenientes del algebra relacional fueron ignoradas. Debido a esto, existen ciertas partes del lenguaje en las que se pueden encontrar determinados puntos débiles, donde la falta de ortogonalidad se convierte en uno de los problemas más relevantes. Aunque existen muchas deficiencias que se han ido eliminando durante estas décadas, la rigidez sintactica asociada a las declaraciones **`SELECT`** continua siendo un problema. Por ejemplo, a pesar de que el algebra relacional permite que sus combinadores puedan aparecer en cualquier lugar, SQL podría exigir al programador la reescritura de una expresión algebraica que se considera idiomática (como **`UNION`** `(tabexp1, tabexp2)`) en su equivalente semántico compatible con su estándar (como `(`**`SELECT`** `...` **`FROM`** `...` **`WHERE`** `...)` **`UNION`** `(`**`SELECT`** `...` **`FROM`** `...` **`WHERE`** `...)`). Afortunadamente, [23] proporciona una lista de reglas sintácticas que se pueden utilizar para reescribir cualquier expresión de un lenguaje ordinario de programación funcional impuro en su correspondiente en SQL. Las expresiones de Optica comparten con el algebra relacional el caracter composicional puro de las

expresiones algebraicas; de hecho, también requieren un conjunto de transformaciones antes de poder ser traducidos en consultas SQL. Estas transformaciones no serán llevadas a cabo en las expresiones de Optica, sino a través de un nuevo dominio semántico que se traduce facilmente a SQL y que utilizaremos como intermediario.

$$
\begin{array}{lcl}
\mathcal{T}^{sql}[\mathit{getter}\ \alpha\ \beta] & = & \mathit{Triplet} \to \mathit{Triplet} \\
\mathcal{T}^{sql}[\mathit{affine}\ \alpha\ \beta] & = & \mathit{Triplet} \to \mathit{Triplet} \\
\mathcal{T}^{sql}[\mathit{fold}\ \alpha\ \beta] & = & \mathit{Triplet} \to \mathit{Triplet} \\
\mathcal{T}^{sql}[\alpha \to \mathit{list}\ \beta] & = & (\mathbb{S} \rightsquigarrow \mathbb{S}) \to \mathit{SQL} \\
\mathcal{T}^{sql}[\mathbb{N}] & = & \mathit{Fragment} \\
\mathcal{T}^{sql}[\mathbb{B}] & = & \mathit{Fragment} \\
\mathcal{T}^{sql}[\mathbb{S}] & = & \mathit{Fragment}
\end{array}
$$

Figura 6.18: Dominios semánticos de SQL

De esta manera, los nuevos dominios semánticos definidos por la función $\mathcal{T}^{sql}$ se muestran en la figura 6.18. En primer lugar, los tipos óptica se mapean en endofunciones *Triplet*. Estas tripletas, que actúan como intermediarias entre las expresiones de ópticas y las expresiones de SQL, tienen como principal propósito el reconciliar los desacuerdos existentes entre ellas. En segundo lugar, el dominio semántico asociado a los tipos consulta se corresponde con una expresión SQL, aunque es necesario proporcionar una función $(\mathbb{S} \to \mathbb{S})$ que traduzca tablas en claves primarias, información que no está contemplada por el modelo de ópticas pero que resulta imprescindible para poder generar consultas SQL. Merece la pena destacar también que los tipos consulta asociados a *get* y *preview* son ignorados. Más tarde se explicará por qué es necesaria la introducción de esta parcialidad. Por último, los tipos base se traducen en *fragmentos de tripleta*, es decir, el resultado de la evaluación de expresiones que denoten estos tipos se usará para formar partes o fragmentos de las tripletas.

Durante el resto de la sección, utilizaremos la misma estructura que viene siendo habitual, donde se introducirá la función $\mathcal{E}^{sql}$ para la evaluación de términos propios del dominio que extienden el lenguaje, términos que denotan ópticas y términos que denotan consultas, y concluiremos debatiendo sobre los resultados obtenidos. Antes de esto vemos imprescindible motivar y describir los detalles sobre la estructura intermedia *Triplet*.

Triplet: motivación y detalles

Como se ha podido observar, una consulta SQL manifiesta una diferenciada separación de aspectos, donde la selección y el filtrado, a pesar de compartir sintaxis, pertenecen a claúsulas diferentes dentro de la consulta. Esta separación hace necesario un mecanismo que nos permita hacer referencia al mismo elemento desde las diferentes cláusulas. SQL aborda este aspecto mediante la declaración de variables en la claúsula `FROM`, accesibles en los ámbitos de las cláusulas `SELECT` y `WHERE`.

Esta manera de representar consultas en SQL contrasta notablemente con respecto a una expresión que utiliza ópticas. Por ejemplo, Optica permite la aparición de combinadores de selección y combinadores de filtrado en cualquier

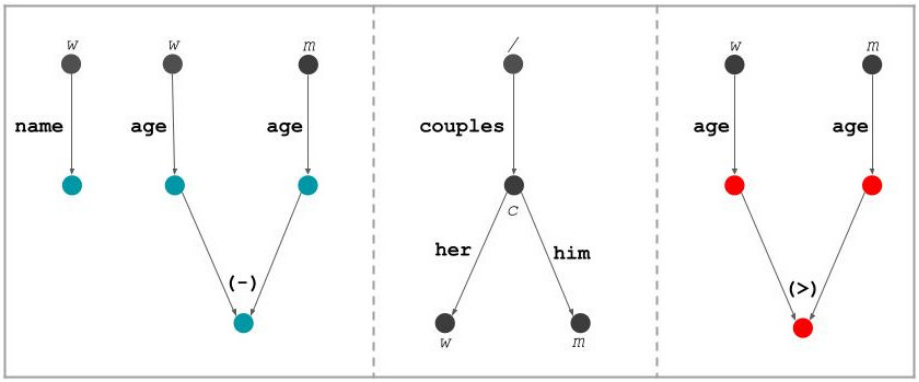

Figura 6.19: Tripleta generada para *differencesFl*.

lugar dentro de la expresión. Además, la información requerida por estos componentes no se nutre de un componente adicional análogo a **FROM**, sino que se especifica bajo demanda. Las expresiones basadas en ópticas tampoco requieren de variables, ya que es el contexto donde dos ópticas aparecen el que determina si realmente están haciendo referencia al mismo elemento o no. Tomaremos la siguiente expresión[5], donde se pueden apreciar dos ocurrencias de *her*:

$$couples \ggg filtered\ (her \ggg age < 50) \ggg her \ggg name$$

A pesar de que una de ellas aparece contenida por el combinador *filtered*, podemos ver que ambos están apuntando a la misma persona. Es importante destacar también que la información requerida por la expresión de filtrado se recoge dentro del ámbito del predicado y no es compartida de manera global.

Triplet es la estructura de datos que utilizamos como intermediaria para conciliar las diferencias descritas en el párrafo anterior. Su principal objetivo es el de separar los tres aspectos fundamentales que son evidentes en una declaración **SELECT** a partir de una expresión de Optica. En particular, una tripleta está formada por tres componentes que se corresponden con las claúsulas **SELECT**, **FROM** y **WHERE**, respectivamente. La figura 6.19 muestra una visión informal de este concepto, donde se presenta la tripleta asociada a la expresión *differencesFl* (definición 31). Podemos ver una tripleta como una óptica estructurada cuyo foco está determinado a partir de sus tres componentes:

- El componente del medio determina la selección potencial de la óptica. Concretamente, la figura 6.19 muestra este componente como un *trie*[6] cuyas aristas son ópticas que seleccionan tipos de entidad (en lugar de tipos base). Sus elementos son secuencias de ópticas que representan una composición vertical de las mismas, por ejemplo la secuencia formada por el fold *couples* y el getter *her* representa el fold $couples \ggg her$. La figura etiqueta cada nodo con un nombre distinto que nos permite referirnos a dicha ruta de forma unívoca. De esta manera, este árbol nos permite apuntar potencialmente a la lista de parejas (c) y a dos listas de personas: las mujeres (w) y hombres (m) que forman parte de una pareja. Los nodos

[5]Esta consulta no es más que una forma menos directa de implementar `under50` (sección 2.5).

[6]`https://en.wikipedia.org/wiki/Trie`

$$
\begin{array}{lcl}
t & ::= & (s,\ f,\ w) \\
s & ::= & (e,\ e,\ \ldots,\ e) \\
f & ::= & / \mid \mathit{insert}\ \hat{p}\ f \\
w & ::= & \{e,\ e,\ \ldots,\ e\} \\
e & ::= & \mathit{like}\ c \mid \mathit{not}\ e \mid e > e \mid e == e \mid e - e \mid \hat{p} \mid \hat{p}.\mathit{optic} \mid \mathit{nonEmpty}\ t \\
\hat{p} & ::= & (\mathit{optic},\ \mathit{optic},\ \ldots,\ \mathit{optic})
\end{array}
$$

Figura 6.20: Sintaxis de Triplet

del trie (de color negro) y sus nombres asociadas pueden ser reutilizadas desde el resto de componentes.

- El componente de la derecha se encarga de restringir las colecciones de entidades que se han identificado en el trie, mediante la imposición de condiciones que sus elementos deben de satisfacer. En el ejemplo, hay una única condición que restringe la colección de parejas (y consecuentemente, las personas que las forman) a aquellas en las que la edad de la mujer (w) supera la edad del hombre (m). Estas condiciones se ilustran en términos de grafos cuyos ejes son ópticas o combinadores binarios como $>$ (que producen la convergencia de dos rutas independientes). Los nodos que conforman estos grafos de restricciones se identifican con una tonalidad roja.

- Por último, pero no por ello menos importante, el componente de la izquierda se encarga de seleccionar las colecciones que formarán parte del resultado final, con la posibilidad de seleccionar y refinar las columnas de tipo base que formarán parte de ésta. En el ejemplo, se selecciona el nombre de la mujer y la diferencia de edad con respecto a su pareja (que será mayor que cero, si se tienen en cuenta las restricciones impuestas por el componente de la derecha). Las selecciones se representan utilizando el mismo tipo de grafos que ya introdujimos en el componente de la derecha, pero simplemente los coloreamos con una tonalidad azul.

La noción de tripleta queda formalizada en la figura 6.20, donde se muestra su sintaxis. Básicamente, una tripleta está formada a partir de tres componentes que se corresponden con las cláusulas **SELECT** (s), **FROM** (f) y **WHERE** (w), respectivamente. Como se ha comentado anteriormente, el componente central es un trie cuyas claves son expresiones de ópticas primitivas que seleccionan entidades. Por tanto, los elementos almacenados en los nodos del trie son secuencias de dichas expresiones, a las que nos referimos como *paths* ($\hat{p}$[7]). Así, un trie podría ser o bien el trie vacío o bien el resultado de insertar un nuevo path en un trie existente $\mathit{insert}\ \hat{p}\ f$.

Los componentes de la izquierda y de la derecha, s y w son una secuencia y un conjunto de expressiones e, respectivamente. Esta distinción se debe a que la repetición de restricciones en w sería redundante y su orden es irrelevante, y por tanto se escoge un conjunto. Las expresiones e son muy similares a las expresiones que pueden encontrarse en Optica (*like*, *not*, $>$, etc.), pero existen ciertos matices diferenciadores que merecen ser explicados. A grandes rasgos,

[7]Se utilizarán *hats*, como en $\hat{p}$, para resaltar los términos que se corresponden con paths.

las expresiones de la tripleta no contemplan la composición vertical como tal. En su lugar, si la composición vertical selecciona una entidad, simplemente se escoge el path correspondiente del trie; si la composición vertical selecciona un tipo base, la expresión se representa como la proyección de un nombre a una óptica que selecciona un tipo base $\hat{p}.optic$. Por ejemplo, la expresión de Optica $couples \ggg her$ denotaría el path $(couples,\ her)$, mientras que la expresión $couples \ggg her \ggg name$ denotaría el path $(couples,\ fst).name$. La composición horizontal tampoco se incluye en esta sintaxis, ya que este aspecto queda cubierto por el componente izquierdo de la tripleta, donde es posible seleccionar una secuencia de expresiones sencillas. Finalmente, la nueva sintaxis incluye un término *nonEmpty*, donde se mantiene una copia del estado de la tripleta, que se podría utilizar para producir consultas anidadas. La sección 6.4.2, donde se formaliza la correspondencia entre tripletas y SQL, mostrará que este termino es finalmente traducido en una operación `EXISTS`.

En este punto, podríamos considerar utilizar *Triplet* como la representación escogida para representar las ópticas en esta infraestructura. Sin embargo, la composición de las tripletas generadas por las subexpresiones podría resultar en una tarea muy laboriosa. En su lugar, nos gustaría contar con una representación que nos ofrezca unas mínimas facilidades bajo esta perspectiva. En este sentido, es más conveniente usar una endofunción de tripletas de manera que cada subexpresión describa el conjunto de cambios que se deben llevar a cabo sobre la tripleta con la que se compondrá verticalmente[8]. Así es como llegamos a la endofunción sobre tripletas $(Trie \to Trie)$, el dominio semántico elegido para representar los tipos óptica. La idea que persigue esta representación se ilustra en la figura 6.21, donde se muestra la evolución descrita por la función generada a partir de la expresión *differencesFl*, que parte de la tripleta vacía, que será formalizada en breve. Los arcos en esta figura se etiquetan con las subexpresiones que denotan ópticas que identifican la transformación que tiene lugar. Como es de esperar, la tripleta final se corresponde con la tripleta que se mostró en la figura 6.19. Las siguientes secciones se apoyarán en esta evolución para motivar las decisiones tomadas en la implementación de $\mathcal{E}^{sql}$.

Evaluación de extensiones del lenguaje

Esta sección introduce la semántica asociada a los términos que extienden el lenguaje de Optica para contemplar elementos propios de dominios particulares, como pueden ser *couples*, *her*, etc., en términos de las transformaciones que estos producen sobre las tripletas de entradas. La formalización puede encontrarse en la figura 6.22, donde se utilizan los términos asociados al dominio de las parejas. Es importante destacar que $\frown$ representa la concatenación de secuencias. Antes de describir la implementación, recurriremos a las figuras 6.21 y 6.23 (donde se muestra el paso *b)* con mayor grado de detalle) para ilustrar los cambios introducidos por cada término de una manera más informal:

- El paso *a)* muestra los cambios introducidos por el término *couples*. Este es un caso muy especial ya que es el que toma la tripleta inicial como

[8] Esto nos evoca la representación funcional de listas basadas en diferencias (`https://wiki.haskell.org/Difference_list`), donde la concatenación se implementa en términos de composición de funciones y la lista se recupera alimentando a la representación propuesta con la lista vacía como entrada. En este caso, los análogos a la lista vacía y a la concatenación son la tripleta vacía (ver definición 33) y la composición vertical.

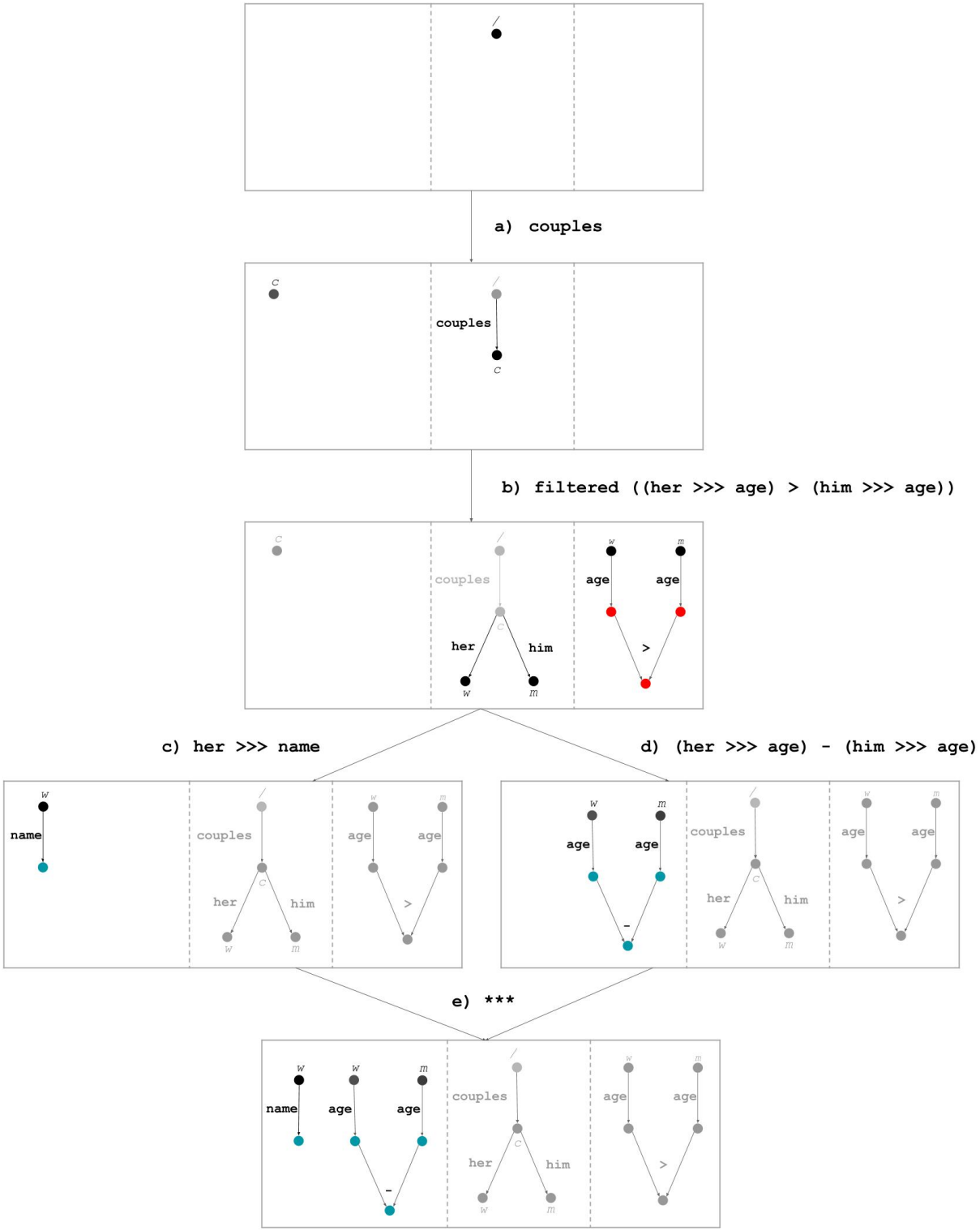

Figura 6.21: Evolución de la tripleta para *differencesFl*.

$$\mathcal{E}^{sql}[\_ : op\ \alpha\ \beta \quad \text{where } op \in \{getter, affine, fold\}] \quad :: \quad Triplet \to Triplet$$

$$\begin{array}{lcl}
\mathcal{E}^{sql}[couples : fold\ Couples\ Couple] & = & entity\ couples \\
\mathcal{E}^{sql}[her : getter\ Couple\ Person] & = & entity\ her \\
\mathcal{E}^{sql}[him : getter\ Couple\ Person] & = & entity\ him \\
\mathcal{E}^{sql}[name : getter\ Person\ \mathbb{S}] & = & base\ name \\
\mathcal{E}^{sql}[age : getter\ Person\ \mathbb{N}] & = & base\ age \\
 & & \\
base\ b & = & ((\hat{x}),\ f,\ w) \mapsto ((\hat{x}.b),\ f,\ w) \quad \text{where} \\
 & & \hat{x} \text{ is an element of } f \\
entity\ e & = & ((\hat{x}),\ f,\ w) \mapsto ((\hat{y}),\ f_2,\ w) \quad \text{where} \\
 & & \hat{y} = \hat{x} \frown (e) \quad \text{and} \\
 & & f_2 = insert\ \hat{y}\ f \quad \text{and} \\
 & & \hat{x} \text{ is an element of } f
\end{array}$$

Figura 6.22: Semánticas no estándar de la extensión para *couples*.

entrada. Como se puede ver, los cambios consisten en la introducción del nuevo path en el trie (componente central de la tripleta) y su selección cruda (componente izquierdo de la tripleta). Se debe tener en cuenta que únicamente es posible introducir paths que apunten a entidades en el trie, como *couples*, que selecciona una secuencia de entidades de tipo *Couple*.

- El paso *b)* contiene más términos propios del dominio de las parejas en el predicado. Si hacemos zoom sobre este paso (figura 6.23) se pueden encontrar los cambios que introduce el término *her*. Concretamente, el paso *b1)* muestra la transformación que este término produce sobre la tripleta generada por *couples*. Teniendo en cuenta que *her* también selecciona una entidad y que se parte de una tripleta no vacía, la tripleta resultante extiende el trie incluyendo la nueva óptica al path introducido previamente para *couples*. Adicionalmente, se reemplaza la selección actual de la óptica, que pasa a convertirse en el nuevo path recién creado (w).
- El paso *b3)* ilustra los cambios introducidos por *age*, donde la novedad reside en que se trata de una óptica que selecciona un tipo base ($\mathbb{N}$) en lugar de un tipo entidad. Consecuentemente, es una expresión que no puede ser introducida en el trie y por tanto lo mantendrá tal cual. Sin embargo, sí que se produce un refinamiento sobre el componente de selección, donde se lleva a cabo la proyección de la nueva óptica sobre un nombre existente en la tripleta de entrada.

Por suerte, el comportamiento de los primeros elementos de la lista puede ser factorizado, siempre y cuando se utilice la siguiente noción de tripleta vacía:

Definición 33. Se formaliza la noción de *tripleta vacía* como aquella que contiene una selección única $\hat{()}$, un trie vacío y un conjunto vacío de restricciones.

$$empty = ((\hat{()}), /, \varnothing)$$

Es importante destacar que la secuencia vacía $\hat{()}$ se usa en el contexto de los tries para hacer referencia a la raíz del mismo, y por tanto, se introduce aquí

para representar el path inicial que aparece en el componente izquierdo (o de selección) de la tripleta.

Sin embargo, resulta inevitable obviar la distinción entre ópticas que seleccionan entidades y aquellas que seleccionan tipos base. La figura 6.22 muestra los métodos *base* y *entity*, que contemplan esta distinción y en los que nos apoyamos para la evaluación de los términos del dominio. La notación $\mapsto$ simplemente representa la notación estándar "maps to" de las funciones. Estos métodos toman la óptica como entrada y producen la endofunción de tripleta correspondiente como resultado. El símbolo $\frown$ representa la concatenación de secuencias. La implementación particular de cada método debería resultar evidente, siempre y cuando se tengan en cuenta las explicaciones previas. No se muestra la evaluación de los términos propios del ejemplo de la organización ya que siguen el mismo patrón, delegando la evaluación a los métodos *base* y *entity*, según proceda.

Evaluación de primitivas *core*

Esta sección precisa las transformaciones que están asociadas a cada uno de los combinadores que forman parte del core de Optica y que pueden encontrarse en la figura 6.24. Antes de adentrarnos en los detalles que subyacen a esta evaluación es necesario introducir varias definiciones, que serán determinantes para asegurar la consistencia de esta formalización:

Definición 34. Dada una expresión $e : \mathit{optic}\ \beta\ \gamma$, donde $\mathit{optic} \in \{\mathit{getter}, \mathit{affine}, \mathit{fold}\}$, una tripleta t es una *entrada válida* para e si se cumple alguna de las siguientes condiciones:

(1) $t = \mathit{empty}$
(2) $t = \mathcal{E}^{sql}[e_2 : \mathit{optic}\ \alpha\ \beta]\ t_2$, para algún e_2, t_2 en el que t_2 es una entrada válida para e_2

Básicamente, una tripleta de entrada es válida para una óptica e si se corresponde con la tripleta vacía (definición 33), o si es el resultado obtenido a partir de la evaluación de otra expresión e_2 que a su vez ha sido alimentada con una entrada válida, donde el tipo que representa la 'parte' de la óptica e_2 debe coincidir con el tipo que representa el 'todo' de la óptica e.

Definición 35. Un *tipo singleton del modelo* es o bien un tipo base o bien un tipo del dominio, es decir, es el resultado de descartar el tipo producto del modelo de tipos de Optica.

Proposición 3. Dado $e : \mathit{optic}\ \alpha\ \beta$, donde $\mathit{optic} \in \{\mathit{getter}, \mathit{affine}, \mathit{fold}\}$, $\beta \in$ tipo singleton del modelo, y t es una entrada válida para e; entonces:

$$((s), \_, \_) = \mathcal{E}^{sql}[e : \mathit{optic}\ \alpha\ \beta]\ t$$

La proposición establece que, dada una entrada válida, el resultado de la evaluación de una óptica que selecciona un tipo singleton siempre devuelve una selección individual s. Esto puede demostrarse fácilmente mediante inducción ya que todos los combinadores que producen tipos singleton configuran una selección individual en el componente izquierdo de la tripleta, de acuerdo con la evaluación mostrada en la figura 6.24. De hecho, esta proposición resulta ser de vital importancia para considerar que la implementación de *base* y *entity* (figura 6.22) están correctamente definidas.

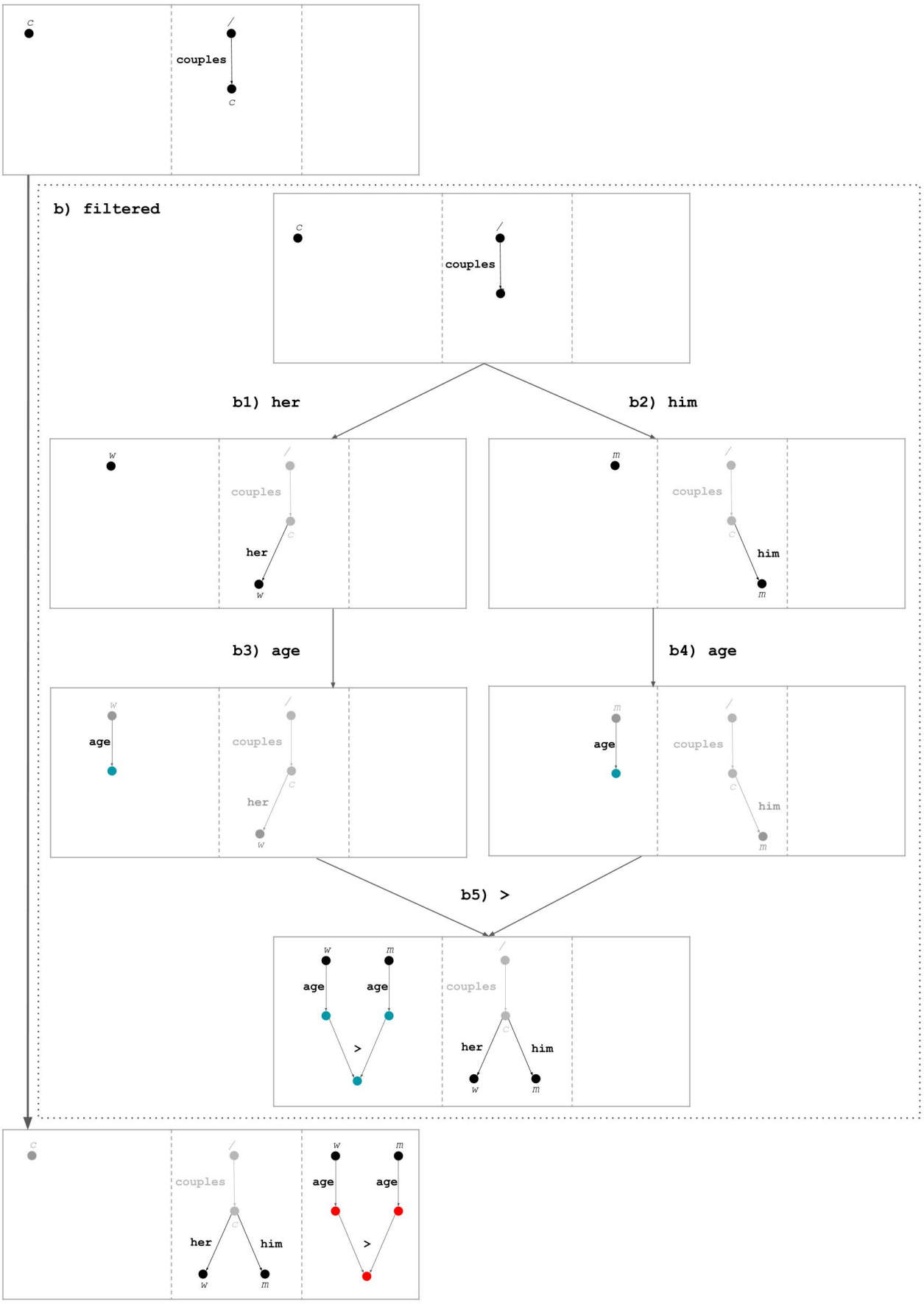

Figura 6.23: Transformación introducida por *filtered* en detalle.

$$\mathcal{E}^{sql}[\_ : op\ \alpha\ \beta \text{ where } op \in \{getter, affine, fold\}] \quad :: \quad Triplet \ \to Triplet$$

$$\begin{array}{lcl}
\mathcal{E}^{sql}[id_{gt} :\ getter\ \alpha\ \alpha] & = & t \mapsto t \\
\mathcal{E}^{sql}[g \ggg_{gt} h : getter\ \alpha\ \gamma] & = & \mathcal{E}^{sql}[h :\ getter\ \beta\ \gamma] \cdot \mathcal{E}^{sql}[g :\ getter\ \alpha\ \beta] \\
\mathcal{E}^{sql}[g\ ***\ h :\ getter\ \alpha\ (\beta,\ \gamma)] & = & t \mapsto (s_1 \frown s_2,\ f_1 \bigtriangledown f_2,\ w_1 \cup w_2) \quad \text{where} \\
 & & (s_1,\ f_1,\ w_1) = \mathcal{E}^{sql}[g :\ getter\ \alpha\ \beta]\ t \quad \text{and} \\
 & & (s_2,\ f_2,\ w_2) = \mathcal{E}^{sql}[h :\ getter\ \alpha\ \gamma]\ t \\
\mathcal{E}^{sql}[like\ b :\ getter\ \alpha\ \beta] & = & (_,\ f,\ w) \mapsto ((like\ _b_),\ f,\ w) \\
\mathcal{E}^{sql}[not\ g :\ getter\ \alpha\ \mathbb{B}] & = & (((b),\ f,\ w) \mapsto ((not\ b),\ f,\ w))) \cdot \mathcal{E}^{sql}[g :\ getter\ \alpha\ \mathbb{B}] \\
\mathcal{E}^{sql}[g\ \oplus h :\ getter\ \alpha\ \delta] & = & t \mapsto ((b_1 \oplus b_2),\ f_1 \bigtriangledown f_2,\ w_1 \cup w_2) \quad \text{where} \\
 & & ((b_1),\ f_1,\ w_1) = \mathcal{E}^{sql}[g : getter\ \alpha\ \beta]\ t \quad \text{and} \\
 & & ((b_2),\ f_2,\ w_2) = \mathcal{E}^{sql}[f :\ getter\ \alpha\ \gamma]\ t \\
 & & \\
\mathcal{E}^{sql}[id_{af} :\ affine\ \alpha\ \alpha] & = & t \mapsto t \\
\mathcal{E}^{sql}[g \ggg_{af} h : affine\ \alpha\ \gamma] & = & \mathcal{E}^{sql}[h :\ affine\ \beta\ \gamma] \cdot \mathcal{E}^{sql}[g :\ affine\ \alpha\ \beta] \\
\mathcal{E}^{sql}[filtered\ g :\ affine\ \alpha\ \alpha] & = & (s,\ f,\ w) \mapsto (s,\ f_1,\ \{b\} \cup w) \quad \text{where} \\
 & & ((b),\ f_1,\ \varnothing) = \mathcal{E}^{sql}[g :\ getter\ \alpha\ \mathbb{B}]\ (s,\ f,\ \varnothing) \\
\mathcal{E}^{sql}[to_{af}\ g :\ affine\ \alpha\ \beta] & = & \mathcal{E}^{sql}[g : getter\ \alpha\ \beta] \\
 & & \\
\mathcal{E}^{sql}[id_{fl} :\ fold\ \alpha\ \alpha] & = & t \mapsto t \\
\mathcal{E}^{sql}[g \ggg_{fl} h : fold\ \alpha\ \gamma] & = & \mathcal{E}^{sql}[h :\ fold\ \beta\ \gamma] \cdot \mathcal{E}^{sql}[g :\ fold\ \alpha\ \beta] \\
\mathcal{E}^{sql}[nonEmpty\ g :\ getter\ \alpha\ \mathbb{B}] & = & (s,\ f,\ w) \mapsto ((nonEmpty\ (\mathcal{E}^{sql}[g :\ fold\ \alpha\ \beta]\ (s,\ f,\ \varnothing)) \\
\mathcal{E}^{sql}[to_{fl}\ a :\ fold\ \alpha\ \beta] & = & \mathcal{E}^{sql}[a : affine\ \alpha\ \beta]
\end{array}$$

Figura 6.24: Semántica no estándar basada en tripletas.

Getters Primero se describirá la implementación de $\ggg_{gt}$. Como la figura 6.21 sugiere, la composición vertical debería ser evaluada como el encadenamiento de transformaciones, es decir, como la composición de funciones. Como consecuencia directa, id_{gt} se implementa como la función identidad, que deja la tripleta tal cual, sin introducir ningún cambio sobre ella.

El paso *e)* (figura 6.21) muestra una ocurrencia de composición horizontal ($***$), donde un par de tripletas divergentes se combinan. En este caso particular, el único componente que sufre alguna modificación es el de selección, ya que los componentes central y derecho son exactamente iguales en ambas tripletas. La evaluación de $***$ proporciona la tripleta de entrada a las funciones resultantes de evaluar g y h, que resulta en la creación de un par de tripletas divergentes, que se corresponden con las que se mostraban en la figura. La combinación de ambas tripletas se lleva a cabo componente a componente. Así, la selección resultante es la concatenación de s_1 y s_2; el trie resultante se produce por la fusión ($\triangledown$) de f_1 y f_2; el componente de restricciones se forma a partir de la unión de los conjuntos w_1 y w_2. Se debe de tener en cuenta que la fusión de un trie y la unión de un conjunto consigo mismos son operaciones idempotentes, y por tanto los componentes central y derecho no introducía ningún cambio en el ejemplo anterior.

A continuación se muestran *like* y *not* como ejemplos de combinadores estándar unarios, que simplemente actualizan el componente de selección de la tripleta de entrada. El primer combinador ignora el componente de selección de la tripleta y simplemente lo reemplaza por la nueva constante. El segundo transforma la tripleta mediante la aplicación de una operación sobre el componente de selección de la entrada. Como se puede apreciar, la evaluación exige que aparezca una expresión individual como parte de la selección, para lo que nos apoyamos en la proposición 3. El sistema de tipos de Optica nos brinda la garantía de que tal expresión representará una óptica que selecciona un booleano.

Finalmente, el paso *b5)* (figura 6.23) representa un combinador binario. La situación es muy similar a la que se introdujo para $***$[9]. Sin embargo, en vez de concatenar las selecciones de ambos componentes, sus selecciones individuales se fusionan en la expresión individual correspondiente. La evaluación de este término asume que las tripletas que derivan de la evaluación de g y h también contienen selecciones individuales. Una vez más, es necesario apoyarse en la proposición 3 ya que todos los combinadores binarios que encontramos en Optica reciben expresiones que denotan tipos base como operandos.

Affine Folds Al igual que ocurría en la evaluación para XQuery (sección 6.3.2), la evaluación de los términos que introducen la composición vertical y la identidad son exactamente los mismos para getters, affine folds y folds, lo cual también queda reflejado por las implementaciones de to_{af} y to_{fl}, que simplemente devuelven la evaluación de su parámetro. Por tanto, sólo queda describir *filtered*, del que ya vimos un ejemplo en el paso *b)* (figura 6.21) cuyos pasos internos quedaban reflejados en la figura 6.23. Esta figura muestra una caja interna que describe la evolución de la tripleta especificada por el predicado, que parte transformando la misma tripleta que la recibida por *filtered*. El resto de la evolución que se muestra dentro de la caja ya debería ser trivial. Sin embargo,

[9]Casualmente, este paso es preferible para ilustrar el resultado de fusionar dos tries.

es necesario describir cómo se genera el resultado final del paso *b)* a partir del resultado que se consigue en el paso *b5)*. Informalmente, lo que sucede aquí es que el componente de selección externo no se ve alterado por la introducción de la restricción y por tanto se mantiene tal cual; el componente izquierdo de la tripleta interna representa el predicado, que pasa a formar parte del conjunto de restricciones en la tripleta externa; finalmente, el componente central se mantiene tal cual en este caso particular.

La evaluación de *filtered* en la figura 6.24 formaliza las intuiciones descritas en el párrafo anterior. En primer lugar, la tripleta de entrada para este término se pasa como argumento a la evaluación del predicado, con un ligero matiz, ya que el componente de restricción (que no es relevante en este contexto) se resetea al conjunto vacío. La tripleta generada por el predicado produce un conjunto vacío de restricciones, ya que los getters no tienen capacidad para actualizar el componente de restricciones de la tripleta. La tripleta resultante mantiene la selección tal cual, añade la selección del predicado al conjunto de restricciones de la tripleta de entrada y configura el trie generado por el predicado como trie definitivo, ya que dicho predicado podría haber introducido nuevos nombres que es necesario contemplar.

Folds Finalmente, se presenta la evaluación de *nonEmpty*, que introduce una diferencia significativa con respecto al resto de combinadores: recibe un fold como parámetro. La evaluación de folds es muy delicada ya que podría derivar en la introducción de consultas anidadas en esta infraestructura, tal y como veremos más adelante. Esta es la razón principal por la que se introduce el término *nonEmpty* como parte de la sintaxis de las tripletas (figura 6.20), que basicamente almacena la tripleta interna que resulta de la evaluación del fold alimentada con la tripleta de entrada (tras resetear sus restricciones, que son irrelevantes en este contexto). La tripleta interna podría extender el trie con nuevos nodos, pero no se propagan a la tripleta externa por considerarse privados a su ámbito. Como consecuencia, el trie y las restricciones se mantienen tal cual.

Observación 25. Una expresión de Optica siempre se puede traducir en una endofunción de tripletas, tal y como muestra la implementación total de $\mathcal{E}^{sql}$, donde la proposición 3 ha resultado ser esencial. De hecho, esta evaluación simplemente se encarga de cambiar los diferentes aspectos de sitio para adaptarse a la estructura impuesta por las tripletas. No obstante, la traducción de tripletas en consultas SQL sí que es un proceso parcial, como se verá a continuación.

Consultas específicas y resultados

Las tripletas han sido diseñadas con el objetivo de ser fácilmente traducibles en declaraciones `SELECT`. Este aspecto se puede apreciar en la figura 6.25 donde se compara la tripleta generada para *differencesFl* (definición 31) y la consulta SQL que se ha presentado en los antecedentes (sección 6.4.1) para este mismo ejemplo. La adaptación de los componentes de selección y restricción es bastante directa, pero la generación de la cláusula `FROM` requiere de una explicación más detallada. La formalización de la traducción de tripletas en consultas SQL queda plasmada en la figura 6.26. Lo primero que llama la atención es la ausencia de traducciones para *get* y *preview*, ya que sólo es posible producir una consulta SQL a partir del término *getAll*.

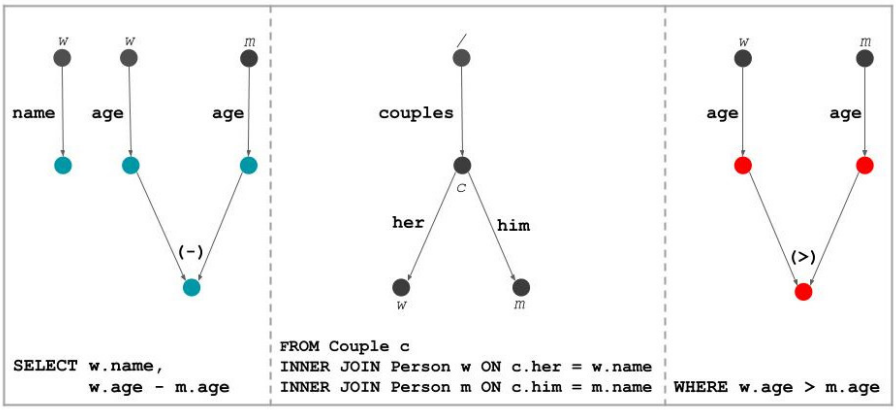

Figura 6.25: From triplet to SQL

Precondiciones 1. Se describen las condiciones precisas que una consulta de Optica debe satisfacer para producir una consulta SQL válida[10]:

1. El tipo que selecciona la óptica, i.e. su 'parte', es un tipo plano. Por ejemplo, *couples* no es traducible a SQL, ya que selecciona una *Couple* como parte, que contiene referencias anidadas a la entidad *Person*. La expresión *couples* ⋙ *her* es válida, ya que *Person* no contiene nuevas entidades anidadas: el nombre y la edad son valores planos.

2. La expresión no puede contener un fold que seleccione un tipo base. Por ejemplo, *departments* ⋙ *employees* ⋙ *tasks* es correcta ya que todos los folds seleccionan tipos entidad.

3. El tipo de óptica original, i.e. ignorando castings, de la expresión que esté más a la izquierda en una consulta tiene que ser un fold. Por ejemplo *couples* ⋙ *her* ⋙ *name* es traducible a SQL, ya que su expresión más a la izquierda es el fold *couples*. Sin embargo, *her* ⋙ *name* no lo es, ya que comienza por el getter *her*. Consecuentemente, *get* y *preview* no se tendrán en cuenta en las traducciones, ya que una expresión que denote un getter o un affine fold no puede satisfacer esta condición.

Cada una de estas condiciones será motivada en los siguientes párrafos, donde se describe el proceso de generación de consultas SQL a partir de tripletas.

Si se pretende traducir una tripleta en una expresión SQL, el primer paso a realizar consiste en producir una tripleta con la que poder trabajar. Esta tarea se lleva a cabo evaluando la expresión fold que acompaña a *getAll* y proporcionando la tripleta vacía (definición 33) a la endofunción resultante. Una vez obtenida la tripleta asociada a la óptica, es necesario refinar su trie extendiendo cada nodo con un nombre *fresh*, ya que serán necesarias durante la traducción. Es importante recordar que los nombres que se asignaban a los diferentes paths del trie eran virtuales y sólo aparecían en los diagramas para facilitar su comprensión, por lo que este paso resulta necesario. Finalmente, se pasa la tripleta

[10]Es importante destacar que se emitirá un error si alguna de las precondiciones no se cumple.

refinada como argumento al traductor final (*sql*). El traductor recibe dos argumentos adicionales: *pk* y $\hat{local}$. El primero de ellos (*pk*) se corresponde con la relación de claves primarias asociadas a cada entidad, información que se le proporciona como parámetro a $\mathcal{E}^{sql}$, tal y como se mostró en la figura 6.18. Por su parte, $\hat{local}$ especifica el path que determina el ámbito de la consulta a generar. Esto es necesario ya que *sql* se usa tanto para producir la consulta SQL total como las consultas anidadas que derivan de la aparición de *nonEmpty* como parte de la expresión. La primera invocación produce la consulta completa; por tanto, se pasa $\rho.\hat{top}$ como argumento, que representa el prefijo común que comparte toda ruta en el trie.

La función *sql* delega la generación de cada cláusula de la declaración **SELECT** al método correspondiente: *select*, *from* y *where*. Además también se produce una llamada a la función $where_+$, cuyo propósito quedará reflejado más adelante. Los resultados obtenidos por cada función son concatenados para formar la consulta final. Merece la pena destacar que los paréntesis y los corchetes son descartados en el resultado, simplemente se usan para delimitar los argumentos que se proporcionan a cada método. En particular, una invocación que aparece rodeada por corchetes está informando de que podría omitirse en determinadas situaciones, donde se tienen en cuenta las condiciones que acompañan a la expresión. Se describirá la generación de cada cláusula en los próximos párrafos, donde se hará referencia a ciertas definiciones recurrentes que describimos justo a continuación:

$\rho.\hat{top}$	*La clave por la que empieza todo path del trie, en caso de que exista*
$\rho_1.local(\rho_2)$	*El path local de ρ_2 por el cual se extiende ρ_1, en caso de que exista*
$\rho(\hat{p})$	*El nombre asignado al path $\hat{p}$ en el trie refinado*
$\hat{p}.last$	*La clave por la que finaliza el path $\hat{p}$*
$\hat{p}.up$	*La ruta que descarta la última clave del path $\hat{p}$*
optic.name	*El nombre asociado a la primitiva optic*
optic.kind	*El tipo de óptica asociado a optic: getter, affine fold o fold*
optic.whole	*El tipo que representa "el todo" asociado a optic*
optic.part	*El tipo que representa "la parte" asociado a optic*
$pk(type)$	*La clave primaria de la tabla relacional asociada al tipo type*

Se abusará de esta notación para omitir la aparición de *last* en algunas expresiones. Por ejemplo se utilizará $\hat{p}.name$ en lugar de la expresión completa $\hat{p}.last.name$.

Cláusula SELECT La función *select* genera la cláusula **SELECT** separando los resultados generados por la traducción de cada expresión en la secuencia mediante comas. La traducción de los diferentes tipos de expresiones queda descrita en las siguientes líneas:

- La traducción del path $\hat{x}$ hace referencia a todas las columnas de la tabla que se corresponden con dicho nombre, que fue previamente asignada por la función *fresh*. Merece la pena destacar que el path debe referirse a una entidad que no contenga referencias a otras entidades anidadas (Precondición 1). De no ser así, la salida de la consulta no recogería toda la información que es necesaria para recrear la entidad; dicho de otra manera, este trabajo no soporta *query shredding* [20].

- La traducción de una proyección $\hat{x}.base$ es muy parecida, aunque en vez de seleccionar todas las columnas, sólo selecciona aquella determinada por la proyección. La implementación muestra una restricción interesante: SQL no ofrece soporte para columnas multivaluadas y por tanto no podemos utilizar un fold como proyección, aunque seleccione un tipo base (Precondición 2).

- La traducción de *nonEmpty* viene dada en términos de una expresión `EXISTS`, que contiene una expresión SQL anidada. Por tanto, se invoca al generador *sql* de forma recursiva. Antes de esto, se necesita, en primer lugar, generar nombres *fresh* para el trie almacenado por *nonEmpty* y fusionar el resultado con el trie externo[11]. En segundo lugar, se necesita calcular el path $\hat{local}$ que será pasado como argumento al generador en la llamada recursiva.

- La evaluación del resto de las expresiones debería ser trivial, ya que simplemente se adaptan operaciones y literales en sus análogos en SQL.

Observación 26. Ninguno de los modelos de ópticas asociados a los ejemplos que guían las explicaciones incluye affine folds en sus definiciones. En el caso particular de la interpretación a SQL, dichas ópticas se asumen como campos que podrían contener el valor `NULL`, i.e. son columnas *nullables* de la tabla.

Cláusula `WHERE` Continuamos por esta cláusula, dada su similaridad con la anterior, que se genera a partir de las funciones *where* y $where_+$. La primera de ellas es muy similar a *select*, ya que básicamente delega en *expr* la evaluación de las expresiones de restricción, aunque en esta ocasión se utiliza `AND` como delimitador para separar los resultados. La evaluación de las expresiones es por tanto idéntica a la que se mostró en el párrafo anterior. En cuanto a $where_+$, cabe destacar que es responsable de añadir las condiciones que conectan las variables externas con las internas, que fueron descritas al final de la sección 6.4.1. A pesar de formar parte de la cláusula `WHERE` la implementación de $where_+$ está muy ligada a la cláusula `FROM` y por tanto su contenido quedará más claro en el próximo párrafo. paragraphCláusula `FROM` Antes de adentrarnos en la función *from*, existen ciertas condiciones que el generador debe de preservar. En primer lugar, se asume que $\rho.\hat{top}$ hace referencia a un fold (Precondición 3), ya que se requiere un punto de entrada en las tablas jerárquicas. Esto significa que únicamente podemos traducir expresiones que comiencen por un fold, como *differencesFl* (definición 31) —que comienza por *couples*— o *expertiseFl* (definición 32) —que comienza por *departments*. En segundo lugar, la invocación a *from* se omite si $\hat{local}$ no está definido, ya que esta situación indicaría que la consulta actual que está siendo generada no introduce nuevas variables y por tanto, no es necesario dotarla de una cláusula `FROM`.

Como era de esperar, la función *from* prepara la cláusula `FROM`. Selecciona el tipo parte asociado a $\hat{local}$ y lo configura como tabla inicial. Después, se encarga de producir una expresión `INNER JOIN` para cada elemento que cuelga de él. Esta es la razón por la que los tries no contienen más que entidades, ya que las

[11] El combinador $\lhd$ fusiona tries, manteniendo los nombres del trie de la izquierda cuando se encuentra con conflictos. Se corresponde con *unionL* de Data.Trie (`http://hackage.haskell.org/package/bytestring-trie-0.2.5.0/docs/Data-Trie.html`)

entidades se corresponden con tablas relacionales. En general, la complejidad de estas definiciones surge por la identificación e implementación del patrón de clave foránea correspondiente (sección 6.4.1).

Una vez que se concluye la implementación de $\mathcal{E}^{sql}$, podemos utilizarla para traducir consultas genéricas en consultas SQL. Para el caso de *differences* (definición 31) obtenemos:

def $differencesSQL : SQL =$

$\mathcal{E}^{sql}[differences : Couples \to list\ (\mathbb{S}, \mathbb{N})]\ (Person \rightsquigarrow name)$

y la adaptación de *expertise* (definición 32) se lleva a cabo de la siguiente forma:

def $expertiseSQL : SQL =$

$\mathcal{E}^{sql}[expertise : Org \to list\ \mathbb{S}]\ (Department \rightsquigarrow dpt,\ Employee \rightsquigarrow emp)$

A diferencia de otros evaluadores, $\mathcal{E}^{sql}$ requiere la relación de claves primarias para las tablas relacionales como un argumento adicional, ya que esta información no se contempla en el modelo de las ópticas. Se usa la notación $(t_0 \rightsquigarrow k_0,\ t_1 \rightsquigarrow k_1,\ ...,\ t_n \rightsquigarrow k_n)$ para construir este argumento. Por ejemplo, la clave primaria asociada a la tabla *Person* es la columna *name*. Si ignoramos los nombres específicos de las variables, las consultas SQL generadas para las definiciones anteriores son exactamente las mismas a las que se introdujeron en la sección 6.4.1.

Como se puede inferir a partir de la figura 6.26, la evaluación de *getAll* lleva a una `SELECT` statement, siempre y cuando no se produzca un error. La consulta resultante no contiene otras consultas anidadas, más allá de las que emergen en el contexto de `EXISTS` (dado el término *nonEmpty*). La claúsula `FROM` usa `INNER JOIN`s como el mecanismo para navegar por las tablas del modelo. Adicionalmente a estos elementos, el evaluador sólo produce expresiones con funciones básicas, operadores y literales: no se requiere ninguna funcionalidad especial de SQL.

Es evidente que la semántica no estándar para la infraestructura de bases de datos relacionales no es tan limpia como la que se propuso para la infraestructura de documentos XML (sección 6.4), ya que esta evaluación ha requerido una normalización en tripletas antes de proceder con la generación de consultas SQL. Además, esta generación es parcial y por tanto las tripletas deben preservar ciertas condiciones para garantizar la correcta traducción a SQL. Afortunadamente, como veremos en la próxima sección, podemos tomar una vía alternativa para la generación de SQL donde podemos apoyarnos en el trabajo existente en el campo de *language-integrated query*. No obstante, más adelante se discutirá por qué esta sección sigue siendo muy importante en el contexto de este trabajo de investigación.

6.5. T-LINQ

Esta sección introduce Optica bajo una nueva perspectiva, en la que se postula como un lenguaje de algo nivel que puede ser interpretado en comprehensions. En concreto, se traducirán las consultas genéricas de Optica en expresiones T-LINQ [19], donde seguiremos una traducción similar a la implementada en el lenguaje Links por [17]. Esta traducción nos demuestra que el estilo composicional atribuido a las ópticas puede ser explotado para generar expresiones basadas

$$
\begin{array}{lcl}
\mathcal{E}^{sql}[getAll\ g:\ \alpha \to list\ \beta]\quad pk & = & sql\ (s,\ \rho,\ w)\ pk\ \rho.\hat{top}\quad \text{where} \\
 & & (s,\ f,\ w) = \mathcal{E}^{sql}[g:\ fold\ \alpha\ \beta]\ empty\quad \text{and} \\
 & & \rho = fresh\ f\quad \text{and} \\
 & & \rho.\hat{top}\ \text{is defined} \\
 & & \\
sql\quad (s,\ \rho,\ w)\quad pk\quad [\hat{local}] & = & (select\ s\ \rho\ pk)\quad [from\ \rho\ pk\ \hat{local}]\quad (where\ w\ \rho\ pk)\quad [where_{+}\ \rho\ pk\ \hat{local}];\quad \text{where} \\
 & & \rho.\hat{top}.kind = fold\quad \text{and} \\
 & & from\ \text{invocation is omitted if}\ \hat{local}\ \text{is not defined}\quad \text{and} \\
 & & where_{+}\ \text{invocation is omitted if}\ \rho.\hat{top} = \hat{local} \\
 & & \\
select\quad (e_1, e_2, \ldots, e_n)\quad \rho\quad pk & = & \textbf{SELECT}\ expr\ e_1\ \rho\ pk\ \textbf{,}\ expr\ e_2\ \rho\ pk\ \textbf{,}\ \ldots\ \textbf{,}\ expr\ e_n\ \rho\ pk \\
 & & \\
expr\quad \hat{x}\quad \rho\quad pk & = & \rho(\hat{x})\ \textbf{.*}\quad \text{where} \\
 & & \hat{x}.last.part\ \in \text{flat types} \\
expr\quad \hat{x}.optic\quad \rho\quad pk & = & \rho(\hat{x})\textbf{.}(optic.name)\quad \text{where} \\
 & & optic.part \in \text{base types}\quad \text{and} \\
 & & optic.kind \in \{getter, affine\} \\
expr\quad (t \oplus u)\quad \rho\quad pk & = & expr\ t\ \rho\ pk \oplus expr\ u\ \rho\ pk \\
expr\quad (not\ e)\quad \rho\quad pk & = & \textbf{NOT (}\ expr\ e\ \rho\ pk\ \textbf{)} \\
expr\quad (like\ a)\quad \rho\quad pk & = & a \\
expr\quad (nonEmpty\ (s,\ f,\ w))\quad \rho\quad pk & = & \textbf{EXISTS (}\ sql\ (s,\ \rho', w)\ pk\ \rho.local(\rho')\ \textbf{)}\quad \text{where} \\
 & & \rho' = \rho \lhd fresh\ f \\
 & & \\
where\quad \varnothing\quad \rho\quad \_ & = & \textbf{WHERE}\ True \\
where\quad \{e_1, e_2, \ldots, e_n\}\quad \rho\quad pk & = & \textbf{WHERE}\ expr\ e_1\ \rho\ pk\ \textbf{AND}\ expr\ e_2\ \rho\ pk\ \textbf{AND}\ \ldots\ \textbf{AND}\ expr\ e_n\ \rho\ pk \\
 & & \\
where_{+}\quad \rho\quad pk\quad \hat{local} & = & \textbf{AND}\ \rho(\hat{local})\textbf{.}key = \rho(\hat{local}.up)\textbf{.}key\quad \text{where} \\
 & & key = pk(\hat{local}.whole) \\
 & & \\
from\quad \rho\quad pk\quad \hat{local} & = & \textbf{FROM}\ \hat{local}.part\ \textbf{AS}\ \rho(\hat{local})\ joins\quad \text{where} \\
 & & joins = \{eqjoin\ \hat{x}\ \rho\ pk \mid \hat{x} \in \rho,\ \hat{x} = \hat{local} \frown \hat{y}\quad \text{for some}\quad \hat{y} \neq ()\} \\
 & & \\
eqjoin\quad \hat{x}\quad \rho\quad pk & = & \textbf{INNER JOIN}\ \hat{x}.part\ \textbf{AS}\ \rho(\hat{x})\ cond\quad \text{where} \\
 & & cond = \begin{cases} \textbf{USING}\ pk(\hat{x}.whole) & \text{if}\quad \hat{x}.kind\ = fold \\ \textbf{ON}\ \rho(\hat{x}.up)\textbf{.}(\hat{x}.up.name) = \rho(\hat{x})\textbf{.}(pk(\hat{x}.part)) & \text{otherwise} \end{cases}
\end{array}
$$

Figura 6.26: Generación de SQL a partir de tripleta.

en comprehensions de forma automática. Además, se evita la tediosa tarea de generar expresiones SQL a partir de expresiones basadas en ópticas, descrita en la sección 6.4, apoyándonos en las técnicas de normalización y traducción existentes en T-LINQ. Como ya es habitual, primero se introducirá brevemente el nuevo dominio de interpretación y después se mostrarán las semánticas no estándar que se requieren para evaluar consultas genéricas en consultas basadas en comprehensions.

$$
\begin{aligned}
&\textbf{type } \mathit{NestedOrg} = \mathit{NestedDepartment}\ \textbf{list}\\
&\textbf{type } \mathit{NestedDepartment} =\\
&\quad \{\mathit{dpt} : \textbf{string};\ \mathit{employees} : \mathit{NestedEmployee}\ \textbf{list}\}\\
&\textbf{type } \mathit{NestedEmployee} =\\
&\quad \{\mathit{emp} : \textbf{string};\ \mathit{tasks} : \mathit{Task}\ \textbf{list}\}\\
&\textbf{type } \mathit{Task} = \{\mathit{tsk} : \textbf{string}\}
\end{aligned}
$$

Figura 6.27: Modelo anidado de organización

$$
\begin{aligned}
\textbf{type } \mathit{Org} = \{&\mathit{departments} : \{\mathit{dpt} : \textbf{string}\}\ \textbf{list};\\
&\mathit{employees} : \{\mathit{dpt} : \textbf{string};\ \mathit{emp} : \textbf{string}\}\ \textbf{list};\\
&\mathit{tasks} : \{\mathit{emp} : \textbf{string};\ \mathit{tsk} : \textbf{string}\}\ \textbf{list}\}
\end{aligned}
$$

Figura 6.28: Modelo relacional de organización

6.5.1. Antecedentes

Esta sección introduce T-LINQ implementando manualmente la consulta análoga a *expertise* (definición 32) en este lenguaje[12]. Antes de nada, se ha considerado conveniente introducir la diferencia entre un modelo anidado y un modelo relacional (o plano). La figura 6.27 muestra el modelo anidado asociado al ejemplo de la organización (*NestedOrg*); por su parte, la figura 6.28 muestra el modelo relacional (*Org*), ambas definiciones extraídas de [19]. La principal diferencia entre ambos tipos de datos reside en que el segundo de ellos despliega claves textuales para referirse al resto de entidades, en lugar de incluir la entidad al completo de manera directa, con lo que este tipo de datos guarda una enorme correspondencia con las tablas relacionales que se introdujeron en la sección 6.4.1.

Cheney *et al* muestran la expresión *quoted* que permite traducir un modelo relacional en un modelo anidado (figura 6.29), donde $\%org$ se corresponde con el *splice* de la expresión que identifica la base de datos (<@**database**("Org")@>). Esta expresión nos permite vislumbrar el estilo asociado a T-LINQ, donde el programador puede trabajar con la base de datos como si se tratara de una simple lista en memoria que trabaja con estructuras de datos provenientes del

[12]Se omitirá el ejemplo de las parejas por brevedad. Se ha seleccionado *expertise* en lugar de *differences* por considerarse una traducción más ambiciosa.

modelo relacional; por tanto, puede usar la notación de comprehensions, muy extendida en el paradigma de programación funcional, para implementar las consultas deseadas, donde existen mecanismos de filtrado (*if...then*) y mapeo (*yield*) a su disposición. La figura 6.30 muestra la implementación de la consulta del ejemplo de la organización en términos de T-LINQ, que también trabaja con el modelo relacional[13]. Más tarde, se mostrará que el modelo anidado se vuelve imprescindible en la evaluación de expresiones de Optica, cuando tratemos de generar automáticamente el equivalente a *expertiseTlinq* a partir de la consulta genérica *expertise*.

```
def nestedOrg = <@
  for d in %org.departments do
  yield {dpt = d.dpt, employees =
           for e in %org.employees do
           if d.dpt = e.dpt then
           yield {emp = e.emp, employees =
                    for t in %org.tasks do
                    if e.emp = t.emp then
                    yield {tsk = t.tsk}}} @>
```

Figura 6.29: Traduciendo el modelo anidado en el modelo plano

```
def expertiseTlinq = <@
  for d in %org.departments do
  if not exists
    for e in %org.employees do
    if d.dpt = e.dpt ∧ not exists
      for t in %org.tasks do
      if e.emp = t.emp ∧ t.tsk = "abstract" then yield t.tsk
    then yield e.emp
  then yield d.dpt @>
```

Figura 6.30: Consulta análoga a *expertise* implementada en T-LINQ

[13]T-LINQ también soporta un estilo composicional, donde se pueden definir combinadores análogos a *all*, *any*, etc. [19, Sección 3.2]. Si se utilizan estos combinadores y se trabaja con la versión anidada del modelo de la organización, la consulta podría ser escrita de forma muy concisa, aunque aquí se prefiere esta versión por motivos didácticos. Ambas versiones producirían la misma consulta SQL, gracias al motor de normalización proporcionado por T-LINQ.

6.5.2. Semántica no estándar

Al igual que en secciones previas, proporcionaremos una función $\mathcal{E}^{tlinq}$ para evaluar expresiones de Optica en expresiones T-LINQ. Antes de esto, es necesario determinar cuáles van a ser los dominios semánticos para esta evaluación, que quedan recogidos mediante la función $\mathcal{T}^{tlinq}$, que mostramos en la figura 6.31. Esta función semántica mapea tipos de Optica en representaciones de tipos T-LINQ. En concreto, la implementación se apoya en una función $\mathcal{T}^{aux}$ y simplemente envuelve el resultado proporcionado por ésta con *Expr*. La implementación de $\mathcal{T}^{aux}$ es directa para tipos base, mientras que las tuplas se mapean en registros. La adaptación de tipos consulta es directa, ya que T-LINQ soporta tipos función de forma nativa, aunque es necesario mapear el tipo *option* en *list*, ya que el primer tipo no tiene un análogo en T-LINQ. Finalmente, los tipos óptica se representan por el tipo consulta que tienen asociado. Las próximas secciones presentan los dominios semánticos asociados a los tipos extendidos por la organización, la implementación de $\mathcal{E}^{tlinq}$ y la discusión sobre las consultas específicas resultantes de la evaluación de consultas genéricas.

$$
\begin{array}{lcl}
\mathcal{T}^{tlinq}[t] & = & \mathit{Expr}{<}\ \mathcal{T}^{aux}[t]\ > \\
 & & \\
\mathcal{T}^{aux}[\mathbb{N}] & = & \textbf{int} \\
\mathcal{T}^{aux}[\mathbb{B}] & = & \textbf{bool} \\
\mathcal{T}^{aux}[\mathbb{S}] & = & \textbf{string} \\
\mathcal{T}^{aux}[(\alpha,\beta)] & = & \{\_1 : \mathcal{T}^{aux}[\alpha], \_2 : \mathcal{T}^{aux}[\beta]\} \\
\mathcal{T}^{aux}[\alpha \rightarrow\ \beta] & = & \mathcal{T}^{aux}[\alpha] \rightarrow\ \mathcal{T}^{aux}[\beta] \\
\mathcal{T}^{aux}[\alpha \rightarrow \mathit{option}\ \beta] & = & \mathcal{T}^{aux}[\alpha] \rightarrow\ \textbf{list}\ \mathcal{T}^{aux}[\beta] \\
\mathcal{T}^{aux}[\alpha \rightarrow \mathit{list}\ \beta] & = & \mathcal{T}^{aux}[\alpha] \rightarrow\ \textbf{list}\ \mathcal{T}^{aux}[\beta] \\
\mathcal{T}^{aux}[\mathit{getter}\ \alpha\ \beta] & = & \mathcal{T}^{aux}[\alpha \rightarrow\ \beta] \\
\mathcal{T}^{aux}[\mathit{affine}\ \alpha\ \beta] & = & \mathcal{T}^{aux}[\alpha \rightarrow \mathit{option}\ \beta] \\
\mathcal{T}^{aux}[\mathit{fold}\ \alpha\ \beta] & = & \mathcal{T}^{aux}[\alpha \rightarrow \mathit{list}\ \beta]
\end{array}
$$

Figura 6.31: Dominios semánticos para evaluación T-LINQ

Evaluación de primitivas *core* y extendidas

Esta sección introduce la evaluación de los tipos y términos que extienden Optica para dar cabida a aspectos propios del dominio de la organización. Como ya se ha visto, los términos propios del dominio introducen ópticas y por tanto deben ser adaptados como expresiones quoted que denotan funciones. En la figura 6.32 se muestran los dominios semánticos asociados a los tipos propios de la organización (donde se extiende la función $\mathcal{T}^{linq\prime}$), donde puede observarse que los tipos se mapean a los correspondientes tipos del modelo anidado (y no del modelo relacional). Este aspecto tendrá relevancia más adelante. La figura también muestra la evaluación de los términos, donde se puede apreciar que la implementación es básicamente una adaptación a T-LINQ del código que se presentó en `OrgModel` (sección 2.1.4) donde se usaron expresiones lambda para construir ópticas concretas.

La evaluación de los combinadores *core* (figura 6.33) también guarda una fuerte correspondencia con la implementación de los combinadores de ópticas en el plano concreto que se vieron en la sección 2.1.4. La principal diferencia

$$
\begin{array}{lcl}
\mathcal{T}^{aux}[Org] & = & NestedOrg \\
\mathcal{T}^{aux}[Department] & = & NestedDepartment \\
\mathcal{T}^{aux}[Employee] & = & NestedEmployee \\
\mathcal{T}^{aux}[Task] & = & Task \\
 & & \\
\mathcal{E}^{tlinq}[departments: \ fold\ Org\ Department] & = & <@\ \mathbf{fun}(ds) \rightarrow ds\ @> \\
\mathcal{E}^{tlinq}[dpt : getter\ Department\ \mathbb{S}] & = & <@\ \mathbf{fun}(d) \rightarrow d.dpt\ @> \\
\mathcal{E}^{tlinq}[employees : fold\ Department\ Employee] & = & <@\ \mathbf{fun}(es) \rightarrow es\ @> \\
\mathcal{E}^{tlinq}[emp : getter\ Employee\ \mathbb{S}] & = & <@\ \mathbf{fun}(e) \rightarrow e.emp\ @> \\
\mathcal{E}^{tlinq}[tasks : fold\ Employee\ Task] & = & <@\ \mathbf{fun}(ts) \rightarrow ts\ @> \\
\mathcal{E}^{tlinq}[tsk : getter\ Task\ \mathbb{S}] & = & <@\ \mathbf{fun}(t) \rightarrow t.tsk\ @>
\end{array}
$$

Figura 6.32: Dominios semánticos y evaluación de términos propios del ejemplo de la organización.

reside en el hecho de que las ópticas concretas se construyen directamente sobre el sistema de tipos de Scala y esta evaluación se hace sobre el sistema de tipos de T-LINQ. Por ejemplo, la evaluación de $***$ crea una expresión lambda que usa registros de T-LINQ en lugar de utilizar una expresión lambda y productos propios de Scala. Otro aspecto a tener en cuenta de esta evaluación es la necesidad de hacer *splice* para introducir las expresiones T-LINQ generadas a partir de la evaluación de las subexpresiones de Optica.

Consultas específicas y resultados

El último paso hacia la generación de las consultas específicas pasa por proporcionar la semántica no estándar para los términos que denotan consultas, que se muestra en la figura 6.34. Este paso resulta trivial ya que las ópticas comparten el mismo dominio semántico que las consultas asociadas a éstas, con lo que únicamente hace falta evaluar el argumento de *get*, *preview* y *getAll*. Sin embargo, todavía existe un pequeño desajuste que se debe tener en cuenta para poder producir las consultas deseadas: las expresiones T-LINQ que se generan mediante $\mathcal{E}^{tlinq}$ trabajan sobre entidades del modelo anidado, como se pudo apreciar en la figura 6.32, a diferencia de la consulta *expertiseTlinq* que hemos mostrado en la sección 6.5.1. Para resolver este conflicto, es necesario reconciliar el modelo relacional con el modelo anidado, para lo que es posible utilizar *nestedOrg* (figura 6.29). Por tanto, simplemente hay que pasar el modelo anidado a la expresión T-LINQ generada mediante $\mathcal{E}^{tlinq}$:

$$\mathbf{def}\ expertiseTlinq = <@\ \%\mathcal{E}^{tlinq}[expertise : Org \rightarrow list\ \mathbb{S}]\ \%nestedOrg\ @>$$

Esta definición se presenta como una versión alternativa a la consulta que fue presentada en los antecedentes, mucho menos eficiente y con una legibilidad que se ve notablemente reducida, todo esto derivado por la complejidad introducida por *nestedOrg*. Por suerte, esto no supone ningún problema, ya que ambas consultas comparten la misma forma normal y por consiguiente, producirán la misma consulta SQL.

$$
\begin{array}{lcl}
\mathcal{E}^{tlinq}[id_{gt} : \ getter\ \alpha\ \alpha] & = & <@\ \mathbf{fun}(a) \rightarrow a\ @> \\
\mathcal{E}^{tlinq}[g\ \ggg_{gt} h : getter\ \alpha\ \gamma] & = & <@\ \mathbf{fun}(a) \rightarrow\ \%\mathcal{E}^{tlinq}[h : getter\ \beta\ \gamma]\ (\ \%\mathcal{E}^{tlinq}[g : getter\ \alpha\ \beta]\ a)\ @> \\
\mathcal{E}^{tlinq}[g\ ***\ h : \ getter\ \alpha\ (\beta,\ \gamma)] & = & <@\ \mathbf{fun}(a) \rightarrow \{\_1 = \%\mathcal{E}^{tlinq}[g : getter\ \alpha\ \beta]\ a,\ \_2 = \%\mathcal{E}^{tlinq}[h : getter\ \alpha\ \gamma]\ a\}\ @ \\
\mathcal{E}^{tlinq}[like\ b : \ getter\ \alpha\ \beta] & = & <@\ \mathbf{fun}(a) \rightarrow b\ @> \\
\mathcal{E}^{tlinq}[not\ g : \ getter\ \alpha\ \mathbb{B}] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{not}\ (\ \%\mathcal{E}^{tlinq}[g : \ getter\ \alpha\ \mathbb{B}]\ a)\ @> \\
\mathcal{E}^{tlinq}[g \oplus h : \ getter\ \alpha\ \delta] & = & <@\ \mathbf{fun}(a) \rightarrow (\ \%\mathcal{E}^{tlinq}[g : \ getter\ \alpha\ \beta]\ a \quad \oplus\ \%\mathcal{E}^{tlinq}[h : \ getter\ \alpha\ \gamma]\ a)\ @> \\
 & & \\
\mathcal{E}^{tlinq}[id_{af} : \ affine\ \alpha\ \alpha] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{yield}\ a\ @> \\
\mathcal{E}^{tlinq}[g\ \ggg_{af} h : affine\ \alpha\ \gamma] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{for}\ b\ \mathbf{in}\ \%\mathcal{E}^{tlinq}[g : affine\ \alpha\ \beta]\ a\ \mathbf{do} \\
 & & \qquad \mathbf{for}\ c\ \mathbf{in}\ \%\mathcal{E}^{tlinq}[h : affine\ \beta\ \gamma]\ b\ \mathbf{yield}\ c\ @> \\
\mathcal{E}^{tlinq}[filtered\ p : \ affine\ \alpha\ \alpha] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{if}\ \%\mathcal{E}^{tlinq}[p : \ affine\ \alpha\ \mathbb{B}]\ a\ \mathbf{then\ yield}\ a\ @> \\
\mathcal{E}^{tlinq}[to_{af}\ g : \ affine\ \alpha\ \beta] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{yield}\ (\ \%\mathcal{E}^{tlinq}[g : getter\ \alpha\ \beta]\ a)\ @> \\
 & & \\
\mathcal{E}^{tlinq}[id_{fl} : \ fold\ \alpha\ \alpha] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{yield}\ a\ @> \\
\mathcal{E}^{tlinq}[g\ \ggg_{fl} h : fold\ \alpha\ \gamma] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{for}\ b\ \mathbf{in}\ \%\mathcal{E}^{tlinq}[g : fold\ \alpha\ \beta]\ a\ \mathbf{do} \\
 & & \qquad \mathbf{for}\ c\ \mathbf{in}\ \%\mathcal{E}^{tlinq}[h : fold\ \beta\ \gamma]\ b\ \mathbf{yield}\ c\ @> \\
\mathcal{E}^{tlinq}[nonEmpty\ g : \ getter\ \alpha\ \mathbb{B}] & = & <@\ \mathbf{fun}(a) \rightarrow \mathbf{exists}\ (\ \%\mathcal{E}^{tlinq}[g : \ fold\ \alpha\ \beta]\ a)\ @> \\
\mathcal{E}^{tlinq}[to_{fl}\ a : \ fold\ \alpha\ \beta] & = & \mathcal{E}^{tlinq}[a : affine\ \alpha\ \beta]
\end{array}
$$

Figura 6.33: Semántica no estándar para primitivas *core*

$$
\begin{array}{lcl}
\mathcal{E}^{tlinq}[get\ g:\ \alpha \to \beta] & = & \mathcal{E}^{tlinq}[g:\ getter\ \alpha\ \beta] \\
\mathcal{E}^{tlinq}[preview\ a:\ \alpha \to option\ \beta] & = & \mathcal{E}^{tlinq}[a:\ affine\ \alpha\ \beta] \\
\mathcal{E}^{tlinq}[getAll\ f:\ \alpha \to list\ \beta] & = & \mathcal{E}^{tlinq}[f:\ fold\ \alpha\ \beta]
\end{array}
$$

Figura 6.34: Semántica no estándar T-LINQ de términos que denotan consultas

Capítulo 7

S-Optica: implementación de Optica en Scala

Este capítulo desarrolla el **Obj. 4** (sección 1.4), donde se muestra una implementación del lenguaje Optica en Scala, bautizada bajo el nombre de *S-Optica* [52] y que se corresponde con el código fuente presentado como material suplementario en [87]. Las secciones que se recogen en este capítulo se organizan de la siguiente manera:

- Se muestra el *embedding* del lenguaje en Scala siguiendo una aproximación tagless-final [13], recogiendo la sintaxis y sistema de tipos, las extensiones necesarias para dar soporte a las aplicaciones, la codificación de las consultas genéricas y la semántica estándar (sección 7.1).
- Se describen los aspectos principales de las interpretaciones no estándar enfocadas en XQuery (sección 7.2), SQL (sección 7.3) y T-LINQ (sección 7.4).

Es importante destacar que este capítulo se centrará únicamente en aquellos aspectos que son relevantes desde el punto de vista de la implementación en Scala, para evitar que se solape el contenido con el del capítulo 6.

7.1. Sintaxis y sistema de tipos

Como se ha mostrado en la sección 2.4, la sintaxis y el sistema de tipos de un DSL tipado se implementan por medio de una *type constructor class* (ver apéndice A), que se corresponde con la clase de representaciones, o posibles interpretaciones, del DSL en cuestión. Esta clase no tiene que estar definida como un bloque monolítico, sino que podría estar descompuesta en otras clases, donde cada una recoge determinados aspectos del lenguaje. En el caso particular de S-Optica, esta división se ha inspirado en la estructura de ópticas y combinadores que se mostraba a lo largo de la sección 2.1.4 y en la distinción entre tipos óptica y tipos consulta que también se introducía en ella. La figura 7.1 muestra las diversas clases que contienen la sintaxis y la semántica asociada a los combinadores de getters, affines y folds; la figura 7.2 muestra la implementación de las consultas, así como la type class Optica, donde se recopilan

todos los combinadores. A continuación se muestran algunos comentarios sobre la implementación:

- Los combinadores primitivos de los diferentes tipos de ópticas (getters, affines y folds) están definidos en sus módulos correspondientes. Los combinadores derivados, aquellos que están implementados en términos de combinadores primitivos (`any`, `all`, etc.) se definen en la clase `OptiCom` donde también se agregan todos los combinadores anteriores utilizando composición por mixins.

- La definición de los combinadores derivados se beneficia de las mismas facilidades sintácticas que las que se asumían en la sección 2.1.4. De hecho su implementación es idéntica a la que se mostraba en la figura 2.6. La diferencia reside en las distintas signaturas y en las posibles semánticas: mientras que la implementación de la figura 2.6 sólo trabaja para ópticas concretas, la implementación de la figura 7.1 trabaja para cualquier representación `Repr[_]`. De esta manera, sería posible instanciar esta clase para que trabajase con ópticas concretas o con cualquier otra representación estándar como podría ser la *van Laarhoven* o la basada en *profunctors*; también se podría instanciar esta clase para que trabajase con XQuery, TripletFun o T-LINQ, ya que este trabajo de investigación las posiciona como representaciones de ópticas legítimas, idea que se discute con más detalle en el capítulo 8.

- Se utilizan los tipos asociados a las ópticas concretas (`Getter`, `AffineFold`, etc.) en las signaturas de estas type classes. ¿Cómo es posible entonces que se pueda trabajar con cualquier representación? Evidentemente, las signaturas no reciben ni devuelven ópticas concretas, sino sus representaciones. Por ejemplo, el combinador `empty` no recibe un fold concreto como parámetro, sino cualquier valor que se pueda entender como un fold. Por tanto, las ópticas concretas se comportan fundamentalmente como *phantom types* [56], que especifican los dominios semánticos abstractos del lenguaje y colaboran con la definición del sistema de tipos.

- Los tipos consulta de S-Optica se corresponden con la noción de *observación* [114]. En el estilo tagless-final es habitual encontrarse con diferentes constructores de tipos para representar las expresiones del lenguaje y sus observaciones. Por lo tanto, la figura 7.2 utiliza `Repr[_]` y `Obs[_]` como representaciones para ópticas y consultas, respectivamente. Esto sería equivalente a tener dos DSLs diferentes, uno para ópticas y otro para consultas.

- Los tipos base de S-Optica también disfrutan de su propia representación. Tal y como se puede apreciar en la definición del combinador `like`, los valores base se representan utilizando el propio sistema de tipos del lenguaje host, es decir, Scala. De hecho, su representación no es `Repr[_]` ni `Obs[_]`, sino el constructor de tipos `Id[_]`. Esta práctica suele ser habitual en el estilo tagless-final[1].

[1] Es importante destacar que esto es viable siempre y cuando los tipos base del lenguaje también existan en Scala.

- Para evitar la introducción de un método `like` específico para cada uno de los tipos base (`Int`, `String` y `Boolean`), se usa el GADT `Base`, cuyas instancias están marcadas como implícitas, para permitir el uso de la sintaxis de *context bound* propia de Scala. Las definiciones de `elem` (que se apoya en `like`) y `equal` (que se pretende restringir exclusivamente a tipos base) también dependen de esta abstracción.

7.1.1. Extensiones del lenguaje

En el capítulo 6 se mostró que es necesario extender la sintaxis y el sistema de tipos del lenguaje Optica para dar cabida a componentes específicos a un dominio. Citando a Kiselyov [71], "extensibility is the strong suite of the tagless-final embedding"; por lo tanto, esta tarea debería de resultar sencilla. De hecho, sólo es necesario declarar una nueva type class donde se incluya una entrada por cada óptica de dominio existente en el modelo, como se muestra en la figura 7.3, que recoge la extensión del lenguaje correspondiente al ejemplo de las parejas. Esta forma de encapsular el modelo de la aplicación guarda una fuerte correspondencia con el diseño de capas de datos descrito en la sección 3.2.1, donde se ofrece una type class que contiene un conjunto de ópticas que resultan de interés. Los tipos `Couples`, `Person`, etc. son las estructuras de datos inmutables que acompañaban al modelo asociado a las parejas en la sección 2.1.4, que se utilizan aquí como meros phantom types [56] que dan soporte al sistema de tipos del lenguaje.

7.1.2. Consultas genéricas

Una vez que se han definido los combinadores principales de S-Optica, así como sus extensiones, debería de ser posible implementar una consulta genéricas. Para ello es necesario que la definición de ésta incluya dependencias a los módulos necesarios. Por ejemplo, así es como se adapta la expresión *differencesFl* que se recoge en la definición 31:

```
def differencesFl[Repr[_]](implicit
    O: OpticaCom[Repr],
    M: CoupleModel[Repr]): Repr[Fold[Couples, (String, Int)]] =
  couples ⋙ filtered((her ⋙ age) > (him ⋙ age)) ⋙
              (her ⋙ name) *** ((her ⋙ age) - (him ⋙ age))
```

Como se puede apreciar, es necesario incluir `OpticaCom` (combinadores principales) y `CoupleModel` (términos específicos del dominio de las parejas) como dependencias. La signatura indica que el tipo resultado es una representación de un fold. Como se puede apreciar no es necesario incluir ninguna referencia a la representación para las consultas, al contrario de lo que nos encontramos en la adaptación de *differences*:

```
def differences[Repr[_], Obs[_]](implicit
    O: Optica[Repr, Obs],
    M: CoupleModel[Repr]): Obs[Couples => List[(String, Int)]] =
  differencesFl.getAll
```

La definición no sólo aparece parametrizada por el type constructor `Repr[_]` sino también por `Obs[_]`, donde se utiliza para representar su resultado.

```
trait GetterCom[Repr[_]] {
  def id_gt[S]: Repr[Getter[S, S]]
  def andThen_gt[S, A, B](u: Repr[Getter[S, A]],
                          d: Repr[Getter[A, B]]): Repr[Getter[S, B]]
  def fork_gt[S, A, B](l: Repr[Getter[S, A]],
                       r: Repr[Getter[S, B]]): Repr[Getter[S, (A, B)]]
  def like[S, A: Base](a: A): Repr[Getter[S, A]]
  def not[S](b: Repr[Getter[S, Boolean]]): Repr[Getter[S, Boolean]]
  def equal[S, A: Base](x: Repr[Getter[S, A]],
                        y: Repr[Getter[S, A]]): Repr[Getter[S, Boolean]]
  def greaterThan[S](x: Repr[Getter[S, Int]],
                     y: Repr[Getter[S, Int]]): Repr[Getter[S, Boolean]]
  def subtract[S](x: Repr[Getter[S, Int]],
                  y: Repr[Getter[S, Int]]): Repr[Getter[S, Int]]
}

trait AffineFoldCom[Repr[_]] {
  def id_af[S]: Repr[AffineFold[S, S]]
  def andThen_af[S, A, B](u: Repr[AffineFold[S, A]],
                          d: Repr[AffineFold[A, B]]): Repr[AffineFold[S, B]]
  def filtered[S](p: Repr[Getter[S, Boolean]]): Repr[AffineFold[S, S]]
  def as_af[S, A](gt: Repr[Getter[S, A]]): Repr[AffineFold[S, A]]
}

trait FoldCom[Repr[_]] {
  def id_fl[S]: Repr[Fold[S, S]]
  def andThen_fl[S, A, B](u: Repr[Fold[S, A]],
                          d: Repr[Fold[A, B]]): Repr[Fold[S, B]]
  def nonEmpty[S, A](fl: Repr[Fold[S, A]]): Repr[Getter[S, Boolean]]
  def as_fl[S, A](afl: Repr[AffineFold[S, A]]): Repr[Fold[S, A]]
}

trait OpticaCom[Repr[_]] extends GetterCom[Repr]
    with AffineFoldCom[Repr]
    with FoldCom[Repr] {
  def empty[S, A](fl: Repr[Fold[S, A]]): Repr[Getter[S, Boolean]] =
    fl.nonEmpty.not
  def all[S, A](fl: Repr[Fold[S, A]])(
               p:  Repr[Getter[A, Boolean]]): Repr[Getter[S, Boolean]] =
    (fl ⋙ filtered(p.not)).empty
  def any[S, A](fl: Repr[Fold[S, A]])(
               p:  Repr[Getter[A, Boolean]]): Repr[Getter[S, Boolean]] =
    fl.all(p.not).not
  def elem[S, A: Base](fl: Repr[Fold[S, A]])(a: A): Repr[Getter[S, Boolean]] =
    fl.any(id_gt === like(a))
}
```

Figura 7.1: Combinadores de ópticas en S-Optica.

```
trait GetterQuery[Repr[_], Obs[_]] {
  def get[S, A](gt: Repr[Getter[S, A]]): Obs[S => A]
}

trait AffineFoldQuery[Repr[_], Obs[_]] {
  def preview[S, A](af: Repr[AffineFold[S, A]]): Obs[S => Option[A]]
}

trait FoldQuery[Repr[_], Obs[_]] {
  def getAll[S, A](fl: Repr[Fold[S, A]]): Obs[S => List[A]]
}

trait Optica[Repr[_], Obs[_]] extends OpticaCom[Repr]
  with GetterQuery[Repr, Obs]
  with AffineFoldQuery[Repr, Obs]
  with Fold[Repr, Obs]
```

Figura 7.2: Symantics de S-Optica (incluye combinadores y consultas).

```
trait CoupleModel[Repr[_]] {
  def couples: Repr[Fold[Couples, Couple]]
  def her: Repr[Getter[Couple, Person]]
  def him: Repr[Getter[Couple, Person]]
  def name: Repr[Getter[Person, String]]
  def age: Repr[Getter[Person, Int]]
}
```

Figura 7.3: Extensión de lenguaje para ejemplo de las parejas.

7.1.3. Semántica estándar

Como se ha podido apreciar con anterioridad, las type classes en el estilo tagless-final son comunmente denominadas *Symantics*, un híbrido que pretende combinar los términos 'syntax' y 'semantics', con el objetivo de enfatizar que una misma abstracción sirve un doble propósito: la declaración de la type class define la sintaxis y el sistema de tipos del lenguaje, mientras que las instancias de la misma proporcionan su semántica. La semántica estándar del lenguaje no es una excepción, que queda definida como una instancia de las type clases que definen el lenguaje. Sin embargo, se trata de una instancia muy particular, donde los dominios semánticos abstractos se convierten en dominios semánticos estándar: simplemente se usa la *type lambda* identidad (`λ[x => x]`) como representación, tanto para las ópticas como para las consultas. La implementación de cada término se apoya en la implementación análoga en el plano concreto. Se muestra la semántica estándar en la figura 7.4. Merece la pena destacar que el término `R` es un nombre habitual para intérpretes meta-circulares en la aproximación tagless-final. No obstante, en lugar de proporcionar una implementación monolítica, se puede apreciar cómo `R` está formado a partir de las diferentes instancias para las diversas type classes que se han ido introduciendo a lo largo de esta sección. Por último, se muestra la figura 7.5, donde se recoge la instancia asociada a la extensión del lenguaje para el ejemplo de las parejas, donde se sigue el mismo patrón, delegando la implementación en la definición de las ópticas concretas correspondientes.

```
trait RGetterCom extends GetterCom[λ[x => x]] {
  def id_gt[S] = Getter.id
  def andThen_gt[S, A, B](u: Getter[S, A], d: Getter[A, B]) = Getter.andThen(u, d)
  def fork_gt[S, A, B](l: Getter[S, A], r: Getter[S, B]) = Getter.fork(l, r)
  def like[S, A: Base](a: A) = Getter.like(a)
  def not[S](b: Getter[S, Boolean]) = Getter.not(b)
  def eq[S, A: Base](x: Getter[S, A], y: Getter[S, A]) = Getter.eq(x, y)
  def gt[S](x: Getter[S, Int], y: Getter[S, Int]) = Getter.gt(x, y)
  def sub[S](x: Getter[S, Int], y: Getter[S, Int]) = Getter.sub(x, y)
}

trait RAffineFoldCom extends AffineFoldCom[λ[x => x]] {
  def id_af[S] = AffineFold.id
  def andThen_af[S, A, B](u: AffineFold[S, A], d: AffineFold[A, B]) = AffineFold.andThen(u, d)
  def filtered[S](p: Getter[S, Boolean]) = AffineFold.filtered(p)
  def as_af[S, A](gt: Getter[S, A]) = gt
}

trait RFoldCom extends FoldCom[λ[x => x]] {
  def id_fl[S] = Fold.id
  def andThen_fl[S, A, B](u: Fold[S, A], d: Fold[A, B]) = Fold.andThen(u, d)
  def nonEmpty[S, A](fl: Fold[S, A]) = fl.nonEmpty
  def as_fl[S, A](afl: AffineFold[S, A]) = afl
}

trait RGetterQuery extends GetterQuery[λ[x => x], λ[x => x]] {
  def get[S, A](gt: Getter[S, A]) = gt.get
}

trait RAffineFoldQuery extends AffineFoldQuery[λ[x => x], λ[x => x]] {
  def preview[S, A](af: AffineFold[S, A]) = af.preview
}

trait RFoldQuery extends FoldQuery[λ[x => x], λ[x => x]] {
  def getAll[S, A](fl: Fold[S, A]) = fl.getAll
}

implicit object R extends Optica[λ[x => x], λ[x => x]]
  with RGetterCom with RGetterQuery
  with RAffineFoldCom with RAffineFoldQuery
  with RFoldCom with RFoldQuery
```

Figura 7.4: Semántica estándar de S-Optica.

```
implicit object CoupleExampleR extends CoupleModel[λ[x => x]] {
  val couples = CoupleModel.couples
  val her = CoupleModel.her
  val him = CoupleModel.him
  val name = CoupleModel.name
  val age = CoupleModel.age
}
```

Figura 7.5: Semántica estándar para extensión del ejemplo de parejas.

Consultas específicas y resultados

Una vez que se ha definido el intérprete que proporciona la semántica estándar de S-Optica es posible utilizarlo para traducir las consultas genéricas en consultas específicas, es decir, funciones que trabajan con estructuras de datos inmutables. Por ejemplo, es posible interpretar `differences` de la siguiente manera:

```
val differencesR: Couples => List[(String, Int)] =
  differences[λ[x => x], λ[x => x]](R, CoupleModelR)
```

Como se puede apreciar, se han específicado tanto los tipos de representación como los intérpretes necesarios de forma manual. Por suerte, podrían ser inferidos por el compilador de forma implícita, tal y como se muestra en esta versión alternativa y más conveniente:

```
val differencesR: Couples => List[(String, Int)] = differences
```

La función resultante es extensionalmente equivalente a la definición `differences` de la sección 2.1.4.

7.2. XQuery

Esta sección presenta la interpretación de XQuery implementada en S-Optica. A pesar de que Scala dispone de sintaxis nativa para dar soporte a XML [100], el lenguaje no contempla utilidades estándar para lidiar con XQuery. De hecho XQuery resulta ser un lenguaje muy extenso [128] del que sólo se han mostrado aquellas utilidades que resultaban necesarias para la formalización del evaluador de expresiones de Optica (sección 6.3). Por eso, se implementará un nuevo tipo de datos que contemple únicamente dichos aspectos de XQuery, que se describirán a continuación. Después se pondrá foco en la semántica no estándar, donde se mostrarán los componentes que forman el intérprete y la generación de las consultas específicas XQuery.

7.2.1. Embedding

La figura 7.6 muestra `XQuery`, el ADT que recoge una simplificación del lenguaje de XQuery. En particular, contiene los principales combinadores de XPath y ofrece soporte para *xml interpolation*. A continuación se describe la correspondencia entre cada uno de los casos del ADT y los elementos que fueron introducidos en la sección 6.3.1:

Document selecciona el nodo *document*, representado mediante `/` en XQuery.

Self hace referencia al eje *self*, representado mediante `.` en XQuery.

Seq permite unificar varios valores XQuery para representar acceso anidado o la aplicación de un filtro, como en `couples/her` o `her[age > 0]`.

Name se utiliza para representar nombres como `couples`, `her` o `age` en el punto anterior.

Filter representa un filtro que alberga un predicado, como en `[age > 0]`.

Func/Oper se utilizan para incluir operaciones unarias y binarias, respectivamente, como `age > 0`.

PInt/PBool/PString permiten representar literales, como `0`.

Element se utiliza para incluir interpolación de xml, como en `her/<tuple>...</tuple>`.

```
sealed abstract class XQuery
case object Document extends XQuery
case object Self extends XQuery
case class Seq(p: XQuery, q: XQuery) extends XQuery
case class Name(s: String) extends XQuery
case class Filter(p: XQuery) extends XQuery
case class Func(op: String, p: XQuery) extends XQuery
case class Oper(op: String, p: XQuery, q: XQuery) extends XQuery
case class PInt(i: Int) extends XQuery
case class PBool(b: Boolean) extends XQuery
case class PString(s: String) extends XQuery
case class Element(tag: String, contents: XQuery*) extends XQuery
```

Figura 7.6: Tipo algebraico de datos para XQuery.

La figura 7.7 muestra la adaptación de la consulta XQuery que se presentó en la sección 6.3.1, que volvemos a incluir aquí para facilitar la comparación entre ambas versiones:

```
/xml/couple[her/age > him/age]/<tuple>
                                <fst>{her/name}</fst>
                                <snd>{her/age - him/age}</snd>
                              </tuple>
```

La versión embebida tiene mucho más ruido, pero hay que tener en cuenta que el programador sólo tendrá que lidiar con el ADT `XQuery` a la hora de implementar el intérprete, que será el encargado de generar consultas como `differencesXQuery` automáticamente. Finalmente y como puede resultar evidente, existe un método que nos permite traducir valores de tipo `XQuery` en consultas XQuery de tipo `String` que serán las que se ejecutarán contra los documentos XML, tal y como ocurre en los tests de la librería[2].

```
val differencesXQuery: XQuery =
  Seq(Document, Seq(Name("xml"), Seq(Name("couple"), Seq(
    Filter(Oper(">", Seq(Name("her"), Name("age")), Seq(Name("him"), Name("age")))),
    Element("tuple",
      Element("fst", Seq(Name("her"), Name("name"))),
      Element("snd", Oper("-", Seq(Name("her"), Name("age")),
                               Seq(Name("him"), Name("age"))))))))))
```

Figura 7.7: Versión de `differences` embebida en Scala/XQuery.

7.2.2. Semántica no estándar

La semántica asociada a los combinadores principales de S-Optica se muestra en la figura 7.8. Uno de los aspectos más relevantes que se muestra aquí es

[2]Para esta tarea se utiliza la librería *BaseX* (`http://basex.org/basex/xquery/`).

la adopción de `λ[x => XQuery]` como representación para las ópticas. La implementación de cada término debería de resultar trivial, ya que simplemente es necesario adaptar los contenidos de la figura 6.15, donde se muestra la evaluación de $\mathcal{E}^{xml}$. En este sentido, uno de los beneficios del estilo tagless-final reside en que los parámetros de los diversos combinadores ya vienen interpretados, por lo que no son necesarias llamadas recursivas para llevar a cabo su evaluación. Por último, la figura 7.9 muestra la instancia del intérprete que contempla los términos extendidos para el ejemplo de las parejas, cuya implementación resulta ser trivial, ya que simplemente adapta los contenidos de la formalización propuesta en la figura 6.14.

```
trait XQueryGetterCom extends GetterCom[λ[x => XQuery]] {
  def id_gt[S] = Self
  def andThen_gt[S, A, B](u: XQuery, d: XQuery) = Seq(u, d)
  def fork_gt[S, A, B](l: XQuery, r: XQuery) = Tuple(l, r)
  def like[S, A](a: A)(implicit B: Base[A]) = B match {
    case IntWitness => PInt(a)
    case StringWitness => PString(a)
    case BooleanWitness => PBool(a)
  }
  def not[S](b: XQuery) = Func("not", b)
  def equal[S, A: Base](x: XQuery, y: XQuery) = Oper("=", x, y)
  def greaterThan[S](x: XQuery, y: XQuery) = Oper(">", x, y)
  def subtract[S](x: XQuery, y: XQuery) = Oper("-", x, y)
}

trait XQueryAffineFoldCom extends AffineFoldCom[λ[x => XQuery]] {
  def id_af[S] = Self
  def andThen_af[S, A, B](u: XQuery, d: XQuery) = Seq(u, d)
  def filtered[S](p: XQuery) = Filter(p)
  def as_afl[S, A](gt: XQuery) = gt
}

trait XQueryFoldCom extends FoldCom[λ[x => XQuery]] {
  def id_fl[S] = Self
  def andThen_fl[S, A, B](u: XQuery, d: XQuery) = Seq(u, d)
  def nonEmpty[S, A](fl: XQuery) = Func("exists", fl)
  def as_fl[S, A](afl: XQuery) = afl
}
```

Figura 7.8: Semántica no estándar XQuery.

```
implicit object XQueryCoupleModel extends CoupleModel[λ[x => XQuery]] {
  val couples = Name("couple")
  val her = Name("her")
  val him = Name("him")
  val name = Name("name")
  val age = Name("age")
}
```

Figura 7.9: Semántica no estándar XQuery para ejemplo de parejas.

```
trait XQueryGetterQuery extends GetterQuery[λ[x => XQuery], λ[x => XQuery]] {
  def get[S, A](gt: XQuery) = Seq(Document, Seq(Name("xml"), gt))
}

trait XQueryAffineFoldQuery extends AffineFoldQuery[λ[x => XQuery], λ[x => XQuery]] {
  def preview[S, A](fl: XQuery) = Seq(Document, Seq(Name("xml"), fl))
}

trait XQueryFoldQuery extends FoldQuery[λ[x => XQuery], λ[x => XQuery]] {
  def getAll[S, A](fl: XQuery) = Seq(Document, Seq(Name("xml"), fl))
}
```

Figura 7.10: Semántica no estándar XQuery para consultas.

Consultas específicas y resultados

Para poder producir consultas XQuery es necesario proporcionar el intérprete para los términos que denotan consultas, cuyos componentes se pueden encontrar en la figura 7.10. Si se tiene en cuenta la observación 23, es natural que la representación para las ópticas sea la misma que la representación para las observaciones (`λ[x => XQuery]`). De nuevo la implementación resulta directa, ya que se apoya en las últimas líneas de la figura 7.8. A continuación se muestra `XQueryOptica`, el intérprete definitivo para XQuery, donde se empaquetan las implementaciones de todos los combinadores de S-Optica que hemos visto a lo largo de la sección:

```
implicit object XQueryOptica extends Optica[λ[x => XQuery], λ[x => XQuery]]
  with XQueryGetterCom with XQueryGetterQuery
  with XQueryAffineFoldCom with XQueryAffineFoldQuery
  with XQueryFoldCom with XQueryFoldQuery
```

Con la definición de `XQueryCoupleModel` y `XQueryOptica`, ya contamos con todos los ingredientes para poder modularizar `differencesXQuery`, utilizando la consulta genérica `differences` como elemento central:

```
val differencesXQuery: XQuery =
  differences[λ[x => XQuery], λ[x => XQuery]](XQueryOptica, XQueryCoupleModel)
```

Una vez más, aunque esta definición resulta más apta para fines didácticos, es preferible dejar trabajar a los mecanismos de inferencia del compilador de Scala:

```
val differencesXQuery: XQuery = differences
```

Destacamos que esta definición devuelve como resultado un valor de `XQuery`, es decir, el tipo algebraico que presentamos al comienzo de esta sección. El programador podría estar interesado en traducir dicho valor en el texto con la XQuery definitiva, funcionalidad ya cubierta por la librería.

7.3. SQL

Esta sección muestra la implementación de SQL por medio de tripletas. Se sigue un estilo muy similar al de la sección anterior, ya que también se definirá un ADT para representar el subconjunto de SQL en el que se centra la traducción. Después se pasará a definir la semántica no estándar y finalmente se presentará cómo modularizar las consultas finales que accederan a las tablas relacionales a partir de las versiones genéricas y las traducciones propuestas.

```
case class SSelect(select: SqlSelect, from: SqlFrom, where: Option[SqlExp])

sealed abstract class SqlFrom
case class SFrom(ts: List[SqlTable]) extends SqlFrom

sealed abstract class SqlTable
case class STable(t: Table, v: Var, js: List[SqlJoin]) extends SqlTable

sealed abstract class SqlJoin
case class SJoin(t: Table, v: Var) extends SqlJoin
case class SEqJoin(t: Table, v: Var, cond: SqlEqJoinCond) extends SqlJoin

sealed abstract class SqlEqJoinCond
case class SOn(l: SProj, r: SProj) extends SqlEqJoinCond
case class SUsing(fn: FieldNme) extends SqlEqJoinCond

sealed abstract class SqlSelect
case class SList(es: List[SField]) extends SqlSelect

sealed abstract class SqlExp
case class SAll(e: String) extends SqlExp
case class SField(e: SqlExp, fn: FieldNme) extends SqlExp
case class SProj(v: Var, fn: FieldNme) extends SqlExp
case class SBinOp(op: String, l: SqlExp, r: SqlExp) extends SqlExp
case class SUnOp(op: String, e: SqlExp) extends SqlExp
case class SCons(v: String) extends SqlExp
case class SExists(sel: SSelect) extends SqlExp
```

Figura 7.11: ADT para subconjunto de gramática SQL.

7.3.1. Embedding

La figura 7.11 muestra el ADT que contiene el subconjunto de SQL que se utilizará. En particular, `SSelect` representa la consulta **SELECT** que será generada por el intérprete. Ésta, a su vez, contiene tres atributos que se corresponden con las claúsulas **SELECT**, **FROM** y **WHERE**, donde este último es opcional, ya que podría ser omitido. Cada uno de estos atributos se describe brevemente en los siguientes puntos:

- El atributo `select` es de tipo `SqlSelect`, que es, esencialmente, una lista de expresiones a las que se les puede asociar un nombre. Este aspecto queda recogido por `SField`, donde el nombre representa el texto que se le asignará a la columna correspondiente en la tabla de resultados. Por su parte las expresiones son variables, proyecciones de columnas sobre variables, literales, etc. Es interesante el caso de `SExists`, que representa una expresión **EXISTS** y en consecuencia contiene una consulta anidada.
- El atributo `from` es de tipo `SqlFrom`, el cual permite representar una tabla con un conjunto de **INNER JOIN**s asociados. Estos últimos simulan los accesos a estructuras anidadas en este contexto relacional.
- Finalmente, el atributo **where** es una expresión SQL opcional. No es necesario contemplar aquí un conjunto de expresiones, ya que el operador `SBinOp` permite conectar dos expresiones con el operador **AND**.

Es importante destacar que este ADT no tipa las expresiones. Por ejemplo, nada

```
type TripletFun = Triplet => Triplet

type Triplet = (List[TExpr], VarTree, Set[TExpr])

sealed abstract class TExpr
case class Path(xs: List[OpticType] = List.empty[OpticType]) extends TExpr
case class Proj(path: Path, ot: OpticType) extends TExpr
case class Like[A](a: A) extends TExpr
case class Not(b: TExpr) extends TExpr
case class Eq(a: TExpr, b: TExpr) extends TExpr
case class GreaterThan(x: TExpr, y: TExpr) extends TExpr
case class Sub(x: TExpr, y: TExpr) extends TExpr
case class NonEmpty(tri: Triplet) extends TExpr

type VarTree = ITree[OpticType, (String, Boolean)]
```

Figura 7.12: Codificación de tripletas.

previene al programador de crear una instancia de una expresión que no denote un `boolean` y utilizarla en el atributo `where`. Esto no resulta un problema, ya que asumiremos que estas instancias son creadas por el intérprete, donde el DSL garantiza un tipado correcto.

7.3.2. Semántica no estándar

La figura 7.12 muestra la codificación en Scala para las tripletas. Al igual que ocurría en la sección 6.4, una tripleta se implementa con una tupla que alberga tres componentes. El primero y el tercero contienen una colección de expresiones de Optica donde se eliminan las primitivas de composición y en su lugar aparece la idea de una proyección. Esto es posible gracias a que el componente intermedio gestiona dichas composiciones y les otorga un nombre que puede ser reutilizado desde las expresiones. El árbol no está implementado mediante un *trie*, sino que utiliza un *rose tree*[3] para su definición, cuyos detalles serán ignorados por brevedad[4]. Finalmente, `TripletFun`, dominio semántico de la interpretación, es un mero alias para una endofunción de tripletas.

Al hilo de lo anterior, los módulos que recogen los diversos combinadores deben instanciarse para `TripletFun`, tal y como se muestra en la figura 7.13, donde se muestra la implementación para algunos de los combinadores. En general, la adaptación de la formalización propuesta en la sección 6.4 es bastante directa, aunque hay un par de aspectos que se deben destacar. En primer lugar, las funciones que hacen referencia a *merge* sirven para fusionar árboles de ópticas, implementados como rose trees. Adicionalmente, existen varios casos de pattern matching que elevan errores `"should␣never␣happen"` que efectivamente nunca deberían de llegar a producirse. Este aspecto es precisamente el que quedaba formalizado en la proposición 3, aunque en el caso de Scala, es necesario incluir estos casos para mantener el proceso de compilación libre de *warnings*.

Por último, la figura 7.14 muestra la interpretación para las primitivas que extienden el modelo de las parejas. Para cada óptica que forma parte del modelo, se utiliza `entity` o `base` (cuya implementación se mostrará en la siguiente

[3] `https://en.wikipedia.org/wiki/Rose_tree`

[4] La implementación del árbol de ópticas basada en tries se recoge como trabajo futuro.

```
trait TripletFunGetterCom extends GetterCom[λ[x => TripletFun]] {
  def id_gt[S] = identity
  def andThen_gt[S, A, B](u: TripletFun, d: TripletFun) = u andThen d
  def fork_gt[S, A, B](l: TripletFun, r: TripletFun) =
    merge3With(l, r)(_ ++ _, _ lmerge _, _ ++ _)
  def like[S, A: Base](a: A) = first.set(List(Like(a)))
  def not[S](b: TripletFun) = b andThen first.modify {
    case List(e) => List(Not(e))
    case _ => throw new Error("should never happen")
  }
  private def greaterThan[S](x: TripletFun, y: TripletFun): TripletFun =
    merge3With(x, y)(
      { case (List(e1), List(e2)) => List(GreaterThan(e1, e2))
        case _ => throw new Error("should never happen") },
      _ lmerge _,
      _ ++ _)
  ...
}
trait TripletFunAffineFoldCom extends AffineFoldCom[λ[x => TripletFun]] {
  def filtered[S](p: TripletFun) = {
    case (s, f, w) => p((s, f, Set.empty)) match {
      case (e, f2, _) => (s, f2, w ++ e)
    }
  }
  ...
}
trait TripletFunFoldCom extends FoldCom[λ[x => TripletFun]] {
  def nonEmpty[S, A](fl: TripletFun) = {
    case (s, f, w) =>
      (List(NonEmpty(fl((s, f.map(x => (x._1, false)), Set.empty)))), f, w)
  }
  ...
}
```

Figura 7.13: Semántica no estándar de `TripletFun`.

```
implicit object TripletFunCoupleModel extends CoupleModel[λ[x => TripletFun]] {
  val couples = entity(FoldType("couples", "Couples", "Couple"), "c")
  val her = entity(GetterType("her", "Couple", "Person"), "w")
  val him = entity(GetterType("him", "Couple", "Person"), "m")
  val name = base(GetterType("name", "Person", "String"))
  val age = base(GetterType("age", "Person", "Int"))
}
```

Figura 7.14: Semántica no estándar de `TripletFun` para ejemplo de parejas.

```
implicit object TripletFunOptica
    extends Optica[λ[x => TripletFun],
                   λ[x => KeyRel => Option[SQL]]]
    with TripletFunGetterCom with TripletFunGetterQuery
    with TripletFunAffineFoldCom with TripletFunAffineFoldQuery
    with TripletFunFoldCom with TripletFunFoldQuery {

  def entity(ot: OpticType, vn: String): TripletFun = {
    case (List(Path(xs)), f, w) => {
      val f2 = index(xs).modify { it =>
        it.copy(forest = it.forest.insertWith((_, e) => e, ot, ITree((vn, true))))
      }(f)
      (List(Path(ot :: xs)), f2, w)
    }
    case _ => throw new Error("should never happen")
  }

  def base(ot: OpticType): TripletFun = first.modify {
    case List(e: Path) => List(Proj(e, ot))
    case _ => throw new Error("should never happen")
  }
}
```

Figura 7.15: Unificación de módulos y definiciones de `entity` y `base`.

sección) en función del tipo que cada óptica selecciona. Esta implementación, por tratarse de una prueba de concepto, no explota las técnicas de metaprogramación de Scala, por lo que la información que es de interés para la evaluación es pasada manualmente como cadenas de texto. Este aspecto será pulido en versiones futuras.

Consultas específicas y resultados

Una vez que ya se han definido las interpretaciones para los combinadores estándar y los específicos para la aplicación, es necesario proporcionar las instancias que permitan generar las consultas SQL finales. A continuación se presenta la instanciación de `FoldQuery`:

```
trait TripletFunFoldQuery
    extends FoldQuery[λ[x => TripletFun], λ[x => KeyRel => Option[SQL]]] {
  def getAll[S, A](fl: TripletFun) = rel => toSql(fl(empty), rel)
}
```

Como se puede apreciar, la representación para las acciones tiene como tipo `λ[x => KeyRel => Option[SQL]]`. Esto indica que la generación de SQL requiere de la relación de tablas y claves primarias para proceder. La parcialidad está determinada con una salida opcional, ya que las claves podrían no ser correctas, aspecto que fue obviado en la sección 6.4 para no introducir ruido adicional en la formalización. La implementación de `getAll` alimenta a la interpretación del fold `fl` con la tripleta vacía (definición 33). Después, delega en `toSql`, método que sabe como traducir una tripleta en una instancia del ADT que se recogía en la figura 7.11, que no se muestra por brevedad.

El *mixin* donde se unifican todas las instancias puede encontrarse en la figura 7.15, donde se aprovecha para incluir las implementaciones para `entity`

y `base`, que lidian con el árbol de ópticas. Gracias a esta instancia, ya es posible traducir la consulta genérica en una consulta SQL:

```
val differencesSQL: KeyRel => Option[SQL] =
  differences[λ[x => TripletFun], λ[x => KeyRel => Option[SQL]]](
    TripletFunOptica, TripletFunCoupleModel)
```

Si el resultado se alimenta con una relación de tablas y claves correcta, se obtendrá la consulta `SELECT` deseada, tal y como muestran los tests que acompañan a la librería.

7.4. T-LINQ

Esta sección muestra la implementación del intérprete T-LINQ de S-Optica. De nuevo será necesario indicar cómo se ha embebido este lenguaje en Scala, aunque en esta ocasión no se utilizará un ADT, sino que también nos valdremos del propio tagless-final para la tarea. Después se procederá de forma habitual, describiendo la semántica no estándar y la modularización de las consultas específicas. No obstante, la implementación de este intérprete en Scala tiene ciertas peculiaridades que requieren ciertos módulos adicionales, que se irán describiendo según sean necesarios.

7.4.1. Embedding

T-LINQ es un lenguaje que surge principalmente para la traducción de expresiones basadas en comprehensions en consultas SQL, tal y como se mostró a lo largo de la sección 6.5. Sin embargo, es evidente que también podría traducirse a consultas sobre estructuras de datos en memoria, lo que se correspondería con su semántica estándar, o incluso en consultas específicas para otras infraestructuras. En definitiva, se trata de un lenguaje que podría tener asociadas múltiples interpretaciones. Por todo esto, también embebemos este lenguaje en Scala adoptando un estilo tagless-final, parametrizado con una representación `Repr[_]`, cuyas symantics se muestran en la figura 7.16. Los términos recogidos en este módulo deberían de ser triviales y quedarán más claros con la adaptación de *expertiseTlinq* (figura 6.30) a la versión embebida. Sí que es conveniente destacar que no existen combinadores para lidiar con registros. En su lugar, se propone `product` para paliar esta carencia.

No obstante, antes de poder proporcionar dicha implementación, es conveniente introducir el modelo relacional en Scala, ya que de él derivará un aspecto importante a tener en cuenta. Las siguientes líneas muestran la adaptación de *Org* (figura 6.28) a Scala:

```
type OrgRel = List[DepartmentRel]
case class DepartmentRel(dpt: String)
case class EmployeeRel(emp: String, dpt_fk: String)
case class TaskRel(tsk: String, emp_fk: String)
```

La problemática surge cuando tratamos de llevar a cabo una operación bastante recurrente al componer expresiones T-LINQ, que se puede apreciar en la figura 6.30. En dicha definición es posible acceder a campos de los registros ($e.emp$, $t.tsk$, etc.), pero la versión embebida de T-LINQ en Scala no sabe cómo recuperar la información análoga, es decir, no podría extraer el nombre de un empleado, de tipo `Repr[String]`, a partir de un valor de tipo `Repr[EmployeeRel]`.

```
trait Tlinq[Repr[_]] {
  def int(i: Int): Repr[Int]
  def bool(b: Boolean): Repr[Boolean]
  def string(s: String): Repr[String]
  def foreach[A, B](
    as: Repr[List[A]])(
    f: Repr[A] => Repr[List[B]]): Repr[List[B]]
  def where[A](c: Repr[Boolean])(as: Repr[List[A]]): Repr[List[A]]
  def yields[A](a: Repr[A]): Repr[List[A]]
  def product[A, B](a: Repr[A], b: Repr[B]): Repr[(A, B)]
  def nil[A]: Repr[List[A]]
  def subtract(x: Repr[Int], y: Repr[Int]): Repr[Int]
  def greaterThan(x: Repr[Int], y: Repr[Int]): Repr[Boolean]
  def equal[A](a1: Repr[A], a2: Repr[A]): Repr[Boolean]
  def and(p: Repr[Boolean], q: Repr[Boolean]): Repr[Boolean]
  def not(p: Repr[Boolean]): Repr[Boolean]
  def exists[A](f: Repr[List[A]]): Repr[Boolean]
  def ifs[A](b: Repr[Boolean], t: Repr[A], e: Repr[A]): Repr[A]
  def lam[A, B](f: Repr[A] => Repr[B]): Repr[A => B]
  def app[A, B](f: Repr[A => B])(a: Repr[A]): Repr[B]
}
```

Figura 7.16: Symantics de T-LINQ.

Para suplir esta carencia, se propone que las consultas tengan una dependencia con la type class que se muestra en la figura 7.17, que recoge el esquema que permitirá acceder a los diversos campos. Para evitar conflicto entre los nombres de estos, se utiliza la extensión *_fk* en aquellas ocasiones en las que el campo se corresponda con una clave foránea. El esquema también incluye primitivas para acceder a las distintas tablas que forman la base de datos de este ejemplo.

Equipados con esta dependencia, debería de ser posible implementar la versión de `expertiseTlinq` embebida en Scala, que se recoge en la figura 7.18. Entre sus parámetros implícitos podemos encontrar referencias a `Tlinq` y `OrgSchema`. La expresión asume la existencia de azúcar sintáctico para invocar a los métodos de `OrgSchema` de manera más idiomática. Por ejemplo, `d.dpt` (donde `d` es de tipo `Repr[DepartmentRel]`) se corresponde con `dpt(d)` y `e.dpt` (donde `e` es de tipo `Repr[EmployeeRel]`) se corresponde con `dpt_fk(e)`. Como se puede apreciar, la consulta resultante guarda una fuerte correspondencia con la que se mostraba en la figura 6.30 que estaba escrita en F#.

```
trait OrgSchema[Repr[_]] {
  def db_department: Repr[List[DepartmentRel]]
  def dpt(d: Repr[DepartmentRel]): Repr[String]
  def db_employee: Repr[List[EmployeeRel]]
  def emp(e: Repr[EmployeeRel]): Repr[String]
  def dpt_fk(e: Repr[EmployeeRel]): Repr[String]
  def db_task: Repr[List[TaskRel]]
  def tsk(t: Repr[TaskRel]): Repr[String]
  def emp_fk(t: Repr[TaskRel]): Repr[String]
}
```

Figura 7.17: Esquema para organización relacional.

```
def expertiseTlinq[Repr[_]](implicit
    Q: Tlinq[Repr],
    S: OrgSchema[Repr]): Repr[List[String]] =
  foreach(db_department)(d =>
  where(not(exists(
    foreach(db_employee)(e =>
    where(d.dpt === e.dpt && not(exists(
      foreach(db_task)(t =>
      where(e.emp === t.emp && t.tsk === string("abstract"))(
      yields(t.tsk))))))(
    yields(e.emp))))))(
  yields(d.dpt)))
```

Figura 7.18: Consulta expertise embebida en Scala/T-LINQ.

7.4.2. Semántica no estándar

Si tenemos en cuenta la implementación de $\mathcal{T}^{tlinq}$ (figura 6.31) los dominios semánticos asociados a los tipos óptica se corresponden en T-LINQ con los tipos consulta que estos denotan. Se debe de encontrar una representación que unifique la traducción de cada óptica al tipo de consulta adecuado. Este es precisamente el propósito de `Wrap` (figura 7.19), un GADT que ofrece un caso para cada tipo de óptica donde cada uno contendrá una expresión T-LINQ con el tipo esperado. La ausencia de un tipo `Option` en T-LINQ nos lleva a utilizar una representación para `preview` que se apoya en `List`, tal y como se puede inferir a partir de $\mathcal{T}^{tlinq}$.

```
sealed abstract class Wrap[Repr[_], A]
case class WrapGetter[Repr[_], S, A](get: Repr[S => A])
    extends Wrap[Repr, Getter[S, A]]
case class WrapAffine[Repr[_], S, A](preview: Repr[S => List[A]])
    extends Wrap[Repr, AffineFold[S, A]]
case class WrapFold[Repr[_], S, A](getAll: Repr[S => List[A]])
    extends Wrap[Repr, Fold[S, A]]
```

Figura 7.19: Dominio semántico de expresiones de ópticas para T-LINQ.

La semántica no estándar queda recogida en la figura 7.20, donde sólo se muestran algunos combinadores, por brevedad, ya que lidiar con `Wrap` introduce un ruido considerable por la necesidad de hacer pattern matching con los parámetros de entrada y por la necesidad de envolver la salida con dicho tipo de datos. Si dejamos estos aspectos de lado, la implementación resulta trivial y consistente con $\mathcal{E}^{tlinq}$ (figura 6.33). El único aspecto que cabría la pena destacar es el uso de `product` (representado con sintaxis infija `***`) por la ausencia de registros, que pone en evidencia lo que ya se contaba en párrafos anteriores.

La interpretación de los términos asociados a la extensión del lenguaje para contemplar elementos del dominio de la organización (`OrgModel`) es bastante sencilla, como puede apreciarse en la figura 7.22. Sin embargo, a diferencia de interpretaciones previas para otras infraestructuras, esta instancia no se puede implementar con un **`object`** ya que está parametrizada por la propia representación de T-LINQ. Además aquí también se hace manifiesta la necesidad de un esquema, pero esta vez para el modelo anidado. Las primitivas que forman el

```
trait TlinqGetterCom[Repr[_]](implicit Q: Tlinq[Repr])
    extends GetterCom[Wrap[Repr, ?]] {
  def fork_gt[S, A, B](
      l: Wrap[Repr, Getter[S, A]],
      r: Wrap[Repr, Getter[S, B]]) = (l, r) match {
    case (WrapGetter(f), WrapGetter(g)) =>
      // Concrete repr.: Getter(s => (l.get(s), r.get(s)))
      WrapGetter(lam(s => app(f)(s) *** app(g)(s)))
  }
  // ...
}
trait TlinqAffineFoldCom[Repr[_]](implicit Q: Tlinq[Repr])
    extends AffineFoldCom[Wrap[Repr, ?]] {
  def andThen_af[S, A, B](
      u: Wrap[Repr, AffineFold[S, A]],
      d: Wrap[Repr, AffineFold[A, B]]) = (u, d) match {
    case (WrapAffine(f), WrapAffine(g)) =>
      WrapAffine(lam(s =>
        foreach(app(f)(s))(a =>
          foreach(app(g)(a))(yields))))
  }
  // ...
}
trait TlinqFoldCom[Repr[_]](implicit Q: Tlinq[Repr])
    extends FoldCom[Wrap[Repr, ?]] {
  def andThen_fl[S, A, B](
      u: Wrap[Repr, Fold[S, A]],
      d: Wrap[Repr, Fold[A, B]]) = (u, d) match {
    case (WrapFold(f), WrapFold(g)) =>
      WrapFold(lam(s =>
        foreach(app(f)(s))(a =>
          foreach(app(g)(a))(yields))))
  }
  // ...
}
```

Figura 7.20: Semántica no estándar T-LINQ.

esquema se muestran en la figura 7.21, para las que también se proporciona azúcar sintáctico en forma de notación postfija, para que las definiciones que dependan de ella puedan adoptar un estilo más natural e idiomático (`_.dpt`, `_.employees`, etc.).

Consultas específicas y resultados

El paso definitivo para completar la instancia es el de proporcionar una implementación para los términos asociados a los tipos consulta: `get`, `preview` y `getAll`. La figura 7.23 recoge la adaptación de la formalización presentada en el capítulo anterior (figura 6.34). Como se puede observar, la única tarea a llevar a cabo es la de extraer el programa T-LINQ que está envuelto por el GADT `Wrap`, la representación escogida para las expresiones que denotan ópticas. Tras instanciar estos módulos, ya es posible componer el intérprete definitivo, que se muestra en la figura 7.24, parametrizado por la representación del programa T-LINQ.

En este punto nos encontramos con el mismo problema que ya se describió en los últimos párrafos de la sección 6.5: nuestro intérprete trabaja con el modelo

```
trait OrgNested[Repr[_]] {
  def Department(dpt: Repr[String], employees: Repr[List[Employee]]): Repr[Department]
  def dpt(d: Repr[Department]): Repr[String]
  def employees(d: Repr[Department]): Repr[List[Employee]]
  def Employee(emp: Repr[String], tasks: Repr[List[Task]]): Repr[Employee]
  def emp(e: Repr[Employee]): Repr[String]
  def tasks(e: Repr[Employee]): Repr[List[Task]]
  def Task(tsk: Repr[String]): Repr[Task]
  def tsk(t: Repr[Task]): Repr[String]
}
```

Figura 7.21: Esquema para organización anidada.

```
implicit def tlinqOrgModel[Repr[_]](implicit
    Q: Tlinq[Repr],
    N: OrgNested[Repr]): OrgModel[Wrap[Repr, ?]] {
  def departments = WrapFold(lam(ds => ds))
  def dpt = WrapGetter(lam(_.dpt))
  def employees = WrapFold(lam(_.employees))
  def tasks = WrapFold(lam(_.tasks))
  def tsk = WrapGetter(lam(_.tsk))
}
```

Figura 7.22: Semántica no estándar T-LINQ para ejemplo de organización.

anidado pero la interacción con la base de datos requiere el modelo relacional. Para paliar este problema se propuso la introducción de *nestedOrg*, cuya adaptación a Scala/T-LINQ se muestra en la figura 7.25. Gracias a esta definición es posible proporcionar la versión alternativa y modularizada de `expertiseTlinq`, que queda descrita en la figura 7.26.

De nuevo, es importante destacar que esta definición está parametrizada por el tipo de representación del programa T-LINQ. Esto significa que se podrían llevar a cabo nuevas traducciones sobre la expresión resultante. De hecho, la traducción de un lenguaje a otro de más bajo nivel es una práctica muy habitual en el paradigma funcional, donde se despliega una arquitectura de capas en la que cada nivel abstrae al programador de los detalles superfluos de las capas subyacentes.

Observación 27. La librería S-Optica [52] contiene un fichero *README* donde se describe brevemente la estructura de la librería y donde se proporcionan enlaces a los diversos intérpretes que se han introducido a lo largo de esta sección. En general, la versión embebida de los lenguajes que intervienen en la librería se pueden encontrar en un subpaquete de `optica`. Los diversos intérpretes que nos permiten traducir un lenguaje en otro se encuentran en el subpaquete `intepreter` del lenguaje origen. Por ejemplo, la versión embebida de XQuery se puede encontrar en el módulo `optica.xquery.XQuery`, mientras que su traducción final a texto se localizará en `optica.xquery.interpreter.ToString`.

```
trait TlinqGetterQuery[Repr[_]] extends GetterQuery[Wrap[Repr, ?], Repr] {
  def get[S, A](gt Wrap[Repr, Getter[S, A]]) = gt match {
    case WrapGetter(g) => g
  }
}

trait TlinqAffineFoldQuery[Repr[_]] extends AffineFoldQuery[Wrap[Repr, ?], Repr] {
  def preview[S, A](af Wrap[Repr, AffineFold[S, A]]) = af match {
    case WrapAffineFold(pr) => pr
  }
}

trait TlinqFoldQuery[Repr[_]] extends FoldQuery[Wrap[Repr, ?], Repr] {
  def getAll[S, A](fl Wrap[Repr, Fold[S, A]]) = fl match {
    case WrapFold(ga) => ga
  }
}
```

Figura 7.23: Semántica no estándar T-LINQ para consultas.

```
implicit def tlinqOptica[Repr[_]](implicit val Q: Tlinq[Repr]) =
  new Optica[Wrap[Repr, ?], Repr]
    with TlinqGetterCom[Repr] with TlinqGetterQuery[Repr]
    with TlinqAffineFoldCom[Repr] with TlinqAffineFoldQuery[Repr]
    with TlinqFoldCom[Repr] with TlinqFoldQuery[Repr]
```

Figura 7.24: Intérprete definitivo de S-Optica para T-LINQ.

```
def nestedOrg[Repr[_]](implicit
    Q: Tlinq[Repr],
    S: OrgSchema[Repr],
    N: OrgNested[Repr]): Repr[Org] =
  foreach(db_department)(d =>
  yields(Department(d.dpt, foreach(db_employee)(e =>
                           where(equal(d.dpt, e.dpt))(
                           yields(Employee(e.emp, foreach(db_task)(t =>
                                                  where(equal(e.emp, t.emp))(
                                                  yields(Task(t.tsk))))))))))
```

Figura 7.25: Adaptación de *nestedOrg* a Scala/T-LINQ.

```
def expertiseTlinqOptica[Repr[_]](implicit
    T: Tlinq[Repr],
    N: OrgNested[Repr]): Repr[Org => List[String]] =
  expertise(tlinqOptica[Repr], tlinqOrgModel[Repr])

def expertiseTlinq[Repr[_]](implicit
    Q: Tlinq[Repr],
    S: OrgSchema[Repr],
    N: OrgNested[Repr]): Repr[List[String]] =
  app(expertiseTlinqOptica)(nestedOrg)
```

Figura 7.26: Modularización de `expertiseTlinq`.

Capítulo 8

Discusión

Este capítulo discute los resultados obtenidos a lo largo de la parte III de esta tesis. La estructura del mismo se organiza de la siguiente manera:

- Se discuten los aspectos generales del lenguaje diseñado (sección 8.1).
- Se comparan los estilos denotados por Optica y por una aproximación basada en comprehensions (sección 8.2)
- Se discute la idoneidad de Optica como alternativa para acceder a fuentes de datos no relacionales (sección 8.3).
- Se discuten las relaciones entre Optica y otras aproximaciones relacionadas tales como ORMs y librerías industriales de LINQ existentes en el ecosistema de Scala (sección 8.4).
- Se compara el estilo de Optica con el de Stateless, donde se enfatizan las características de Optica que le permiten abordar las limitaciones de la aproximación basada en optic algebras (sección 8.5).

8.1. El lenguaje de las ópticas

Uno de los aspectos más relevantes de las ópticas es la *modularidad*, es decir, la capacidad de formar ópticas que lidien con estructuras de datos compuestas a partir de otras ópticas más simples que se definan sobre sus componentes. Esta característica se aprecia perfectamente en el framework de profunctor optics [105], donde la composición de las ópticas se basa en la composición de funciones y posibilita la combinación de isos, prismas, lenses, affine traversals y traversals de una manera muy directa. La representación basada en profuntores es muy conveniente para revelar la estructura composicional de las ópticas, pero es importante destacar que este aspecto también está arraigado en otras representaciones como la concreta, van Laarhoven, etc. La modularidad es propia del *lenguaje* de ópticas, más que de una representación en particular. Este trabajo ha mostrado, aunque para un subconjunto de ópticas (getters, affine folds y folds), que la estructura composicional de las ópticas se puede codificar en el sistema de tipos de un lenguaje formal al que se ha bautizado como Optica. La semántica denotacional de este lenguaje viene dada en términos de ópticas

concretas pero cualquier otra representación isomorfa podría haber servido para la tarea.

La especificación de Optica no incluye únicamente el carácter composicional de las ópticas de sólo lectura, sino también las consultas que estas ópticas denotan (*get*, *preview*, etc.). Se ha hecho mucho empeño en separar estos aspectos, lo cual se hará más evidente cuando se aborde la extensión de Optica con nuevas variedades de ópticas. Por ejemplo, la principal diferencia entre un fold y un traversal no reside en su capacidad de composición, sino en las consultas soportadas: además de *getAll*, un traversal debe soportar consultas de actualización que reemplacen el valor de las partes seleccionadas.

En cuanto a la implementación, se ha mostrado que el estilo typed tagless-final resulta muy conveniente para codificar esta separación entre ópticas y consultas, ya que se corresponde con la diferencia entre *representaciones* y *observaciones*. Otra característica esencial de esta aproximación para el desarrollo de DSLs es su capacidad de extensibilidad, que fue mostrada en la sección 2.4.2. En concreto, se incluirán nuevas ópticas en S-Optica que serán contempladas en sus propias type classes (como ya se ha hecho para getters, affine folds y folds) para que las consultas existentes puedan ser reutilizables sin necesidad de sufrir alteraciones o simplemente sin que se requiera su recompilación.

8.2. Ópticas y comprehensions

Tal y como se ha mostrado en la sección 6.5, las ópticas pueden integrarse perfectamente con las comprehensions. De hecho, el intérprete de T-LINQ permite mezclar libremente expresiones basadas en ópticas con consultas generales sobre comprehensions. En este sentido, se podría decir que las ópticas juegan un papel similar al desempeñado por XPath en conexión con XQuery [17]. Los siguientes párrafos discuten el equilibrio entre expresividad y modularidad de las consultas basadas en comprehensions y las consultas de Optica, para apreciar mejor el rol de cada una en el territorio de LINQ.

La separación de aspectos entre la selección de partes de una estructura de datos de forma *declarativa* y la construcción de *consultas* asociadas a esas partes es uno de los pilares de las ópticas. En este sentido, la aproximación para LINQ basada en comprehensions se centra únicamente en la composición de las consultas, y habitualmente, sólo se centra en un tipo de consultas: consultas de lectura que denotan un multiset (el dominio semántico de consultas en QUEΛ [114], T-LINQ [19], NRC [10], etc.). La aproximación basada en ópticas es potencialmente más modular. Por ejemplo, una misma representación de traversals debería permitir la generación tanto de sentencias `SELECT` como de sentencias `UPDATE` para consultas de lectura y escritura, respectivamente.

Se pueden atribuir ventajas de modularidad adicionales a las ópticas, debidas al hecho de que las ópticas proporcionan un lenguaje más cercano al álgebra relacional que al cálculo relacional en el que se basan las mónadas para proporcionar las comprehensions [38]. Aunque es posible defender que el soporte para la abstracción funcional y estructuras de datos anidadas intermedias en lenguajes y sistemas como Links, T-LINQ o DSH[1] también lleva a consul-

[1] En particular, DSH viene con un amplio catálogo de combinadores de procesamiento de listas: `https://github.com/ulricha/dsh/blob/master/src/Database/DSH/Frontend/Externals.hs`.

tas composicionales[2], es posible encontrar consultas donde la diferencia entre ambos estilos resulta muy evidente. Por ejemplo, esta sería la adaptación de la consulta presentada en la sección 2.5 para Optica:

```
def under50_d[Repr[_], Obs[_]](implicit
    O: Optica[Repr, Obs],
    M: CoupleModel[Repr]): Obs[Couples => List[String]] =
  (couples ⋙ her ⋙ filtered (age < 50) ⋙ name).getAll
```

y esta es la implementación análoga para la versión embebida de T-LINQ:

```
def under50_e[Repr[_]](
    couples: Repr[Couples])(implicit
    Q: Tlinq[Repr],
    N: CoupleNested[Repr]): Repr[List[String]] =
  foreach(couples)(c => where (c.her.age < 50) (yields (c.her.name)))
```

Como se puede apreciar, el hecho de adoptar el lenguaje de ópticas deriva en varios beneficios con respecto a la modularidad. En primer lugar, como ya se ha mencionado con anterioridad, la consulta se compone de dos partes principales: la composición de la óptica, que declara la selección, y la expresión con la consulta, que especifica el tipo de consulta que se debe ejecutar sobre la selección. En segundo lugar, la expresión basada en ópticas no introduce variables, construyéndose sobre módulos reusables de grano fino, como vienen siendo `couples`, `her`, `age` y `name`. Esto resulta en consultas puramente algebraicas que son más simples de componer y mantener. En esencia, las consultas se producen componiendo ópticas a partir de otras más simples en un estilo composicional puro y derivando la consulta asociada a éstas.

La desventaja de la aproximación basada en ópticas con respecto a las comprehensions, al menos en la versión actual de la librería, es su expresividad limitada. De hecho, las variables son explotadas por las comprehensions para expresar *joins* arbitrarios (por ejemplo, cíclicos) mientras que las consultas basadas en ópticas están limitadas a recorrer la jerarquía establecida por la estructura de datos anidada partiendo de la raíz de la misma. Los modelos relacionales son más generales que los modelos anidados, dotando al programador con mejores herramientas de navegación [5], y consecuentemente, no todo modelo se puede expresar con las abstracciones de Optica. Se tomará el modelo de las parejas como ejemplo para ilustrar esta idea. En este sentido, se asume que cada persona cuelga de una pareja, por lo que es posible encontrar a todas las personas recorriendo las ramas `her` y `him`. Sin embargo, el modelo relacional puede contemplar entradas para personas que no necesariamente forman parte de una pareja. Para aliviar esta situación, se podría introducir un nuevo fold `people` en el modelo junto con `couples`, donde ambos compartieran una raíz virtual como fuente. Aún así, las conexiones entre `people` y `her`/`him` seguirían sin estar muy claras en el modelo de las ópticas; por tanto, sería necesario introducir mecanismos adicionales que establecieran las relaciones precisas entre ellos. Una investigación más precisa que compare el lenguaje de las ópticas con el de las comprehensions se deja como trabajo futuro, así como la extensión de otras técnicas como agrupación, agregación y ordenación, estudiadas en el campo de LINQ para comprehensions [114, 73, 68].

[2]De hecho, la definición de la consulta *expertise* en Optica no es más simple que su versión equivalente en T-LINQ donde se utilizan estructuras intermedias anidadas [19].

8.3. Ópticas como un lenguaje de consultas

El rol que desempeñan las ópticas en LINQ va más allá de su aplicación en combinación con las comprehensions. De hecho, embeber el lenguaje de las ópticas en un DSL abre la puerta a interpretaciones no estándar que traducen directamente el lenguaje de las ópticas a consultas que permiten acceder a diversas fuentes de datos, incluyendo pero no restringiéndose a estructuras de datos inmutables. Por ejemplo, se ha presentado una interpretación que permite la traducción de expresiones de Optica en consultas XQuery donde se ha podido apreciar que la conexión entre ambas tecnologías es muy fluida. La traducción ignora la sintaxis FLWOR de XQuery y esencialmente se centra en XPath. De hecho, XPath puede verse como un lenguaje que selecciona elementos de un documento XML, lo que hace que sea un ejemplo perfecto de representación de óptica. Además, teniendo en cuenta que XPath no proporciona mecanismos para actualizar un documento XML, esta tecnología encaja perfectamente con las ópticas de sólo lectura contempladas en Optica.

Es importante destacar que las sinergias entre XML y las ópticas no son nuevas en absoluto. De hecho, las librerías de ópticas más extendidas suelen incluir módulos para lidiar con XML[3] o documentos JSON, a veces incluso empaquetados en forma de DSLs, como es el caso de JsonPath [4]. En estos proyectos, existen ópticas estándar para facilitar la definición de expresiones para la consulta de documentos JSON o XML. Sin embargo, la aproximación de Optica es radicalmente diferente, ya que se postula como un lenguaje de ópticas general que permite la composición de expresiones genéricas que podrían ser traducidas a estos DSLs (JsonPath, XQuery, etc.).

La aproximación de Optica también es diferente a la que se ofrece en [19] para lidiar con XQuery, donde se muestra un flujo de trabajo inverso: una expresión XPath es traducida en una expresión basada en comprehensions que permite en última instancia consultar tablas relacionales. En esta línea, se podría conectar este trabajo con SilkRoute [30], donde el administrador de la base de datos expone los datos contenidos en ésta mediante una vista pública XML y los componentes externos lanzan consultas XQuery contra el framework. Entonces, SilkRoute traduce dichas consultas XQuery en una o varias consultas SQL y recolecta los resultados de la base de datos, que son devueltos como documentos XML. Uno de los aspectos clave de SilkRoute son los *view forests*, concepto que se explota en el framework para separar la estructura XML de su computación. Por su parte, Optica expone la jerarquía del dominio mediante un conjunto de ópticas que los componentes externos utilizan para formar las correspondientes consultas. Además, la idea de view forest podría verse como un tipo de óptica ya que también selecciona partes de la base de datos subyacente. En cualquier caso, Optica es más general, considerando que las mismas consultas se podrían utilizar para atacar diferentes fuentes de datos, sin estar restringido exclusivamente a SQL.

Indiscutiblemente, generar SQL es el principal objetivo del clásico LINQ con comprehensions y por tanto resultaba inevitable que se contemplara también en este trabajo. Habitualmente, las consultas basadas en comprehensions requie-

[3] `https://hackage.haskell.org/package/xml-lens-0.1.6.3/docs/Text-XML-Lens.html`

[4] `https://github.com/julien-truffaut/jsonpath.pres`

ren un proceso de normalización previo para garantizar un buen rendimiento: la traducción directa de comprehensions en SQL no es óptima ya que deriva en subconsultas anidadas. Además, el proceso de traducción de consultas flat-flat en SQL es total y evita el problema de la *avalancha de consultas* [23]. En sistemas como Links o T-LINQ, estas garantías se preservan de forma estática. La sección 6.4.2 muestra cómo la traducción de expresiones de Optica a SQL tiene unas garantías similares en las consultas generadas, carente de subconsultas (más allá de las generadas para `EXISTS` que resultan inevitables). Sin embargo, los errores se detectan en tiempo de ejecución, aspecto que pretende solventarse en versiones futuras aplicando técnicas de optimización ofrecidas por el estilo tagless-final [114]. El proceso de traducción a SQL tiene ciertos parecidos con la aproximación denotacional de SQUR [73], más que con la aproximación basada en reescrituras que se sigue en [23, 19, 114]. En particular, la interpretación hace uso de un lenguaje intermedio basado en tripletas que pretende desacoplar los aspectos de recolección, filtrado y selección que son necesarios para generar la consulta SQL definitiva. La interpretación se desmarca de SQUR en el hecho de que la consulta final no se genera a partir de una expresión normalizada sino directamente a partir de la tripleta. Se pretende incorporar técnicas de normalización y evaluación parcial en trabajo futuro, que será conveniente tan pronto como el lenguaje incluya proyecciones π_1 y π_2 que inviertan el efecto introducido por el combinador $***$.

Dada la traducción de expresiones de Optica en comprehensions mostrada en la sección 6.5, uno podría cuestionarse la utilidad de la traducción a tripletas. En este sentido, se argumenta que la traducción es interesante en sí misma por dos razones. En primer lugar, la interpretación allana el terreno en vistas a contemplar consultas de actualización, es decir, sentencias de tipo `UPDATE`, ya que se prevé que se pueda reutilizar gran parte de la implementación. En segundo lugar, la traducción a SQL mediante tripletas ofrece un ejemplo de intérprete similar al de [73] pero llevado a cabo bajo la configuración algebraica de las ópticas en vez del cálculo relacional de las comprehensions. Este estilo podría tomarse como punto de referencia para otros intérpretes relacionados, como la generación de consultas para bases de datos NoSQL como MongoDB[5].

8.4. Optica, ORMs y librerías de LINQ

La existencia de conexiones entre ópticas y bases de datos relacionales es bien conocida. De hecho, las lenses surgieron en este contexto [33] bajo el paraguas de lo que se conoce como *programación bidireccional*. Se destaca [58] como trabajo reciente en este campo, donde se presenta una aproximación práctica para abordar el *view update problem* mediante las denominadas *incremental relational lenses*. Aunque se desconoce si la extensión de Optica requerirá de vistas en la semántica no estándar de SQL, este trabajo de investigación resulta esencial para lidiar con ópticas de actualización de manera efectiva.

Resulta inevitable relacionar este trabajo de investigación con el correspondiente a los *object-relational mappers* (ORMs), donde destaca Hibernate. Ambas aproximaciones persiguen el mismo propósito: trabajar con almacenes de datos persistentes como si se tratará de simples estructuras de datos en memoria. Sin

[5] `https://www.mongodb.com/`

embargo, S-Optica usa el lenguaje de las ópticas mientras que los ORMs tratan de mantenerse lo más cerca posible del estilo de orientación a objetos. A continuación se ofrece una lista de otras diferencias que se consideran relevantes:

- S-Optica no se centra en bases de datos relacionales como única infraestructura. De hecho, las secciones 6.2.3 y 6.3 dan buena cuenta de que las estructuras de datos en memoria y los documentos XML son también fuentes potenciales de información para los que se pueden generar consultas.

- S-Optica es eminentemente declarativo. De hecho, las consultas genéricas no son más que valores[6] que no producen efectos de lado por sí mismos. Esto contrasta con los ORMs, donde se requiere un gran conocimiento sobre la tecnología particular para descubrir qué consultas se están lanzando en cada momento. El estilo declarativo de S-Optica permite tanto la composicionalidad como la capacidad de introducir numerosas optimizaciones sobre las consultas.

- Las consultas escritas mediante S-Optica son expresivas y están bien tipadas. Muchos ORMs extienden su funcionalidad con lenguajes artificiales para expresar consultas, que suelen codificarse como simples cadenas de texto, por lo que los errores asociados a las mismas no se producen hasta su ejecución.

- Los ORMs consideran la noción de objeto como el grano de modularidad más pequeño, por lo que es necesario recuperar todos los valores que lo forman, lo que puede resultar poco eficiente. Por su parte, S-Optica soporta consultas que pueden centrarse en partes muy específicas de una estructura de datos, sin necesidad de recuperar datos innecesarios.

- Los ORMs permiten la escritura de información en las bases de datos, mientras que S-Optica en su estado actual únicamente soporta lecturas, aunque se planea extender esta funcionalidad en versiones futuras.

Los ORMs llevan un largo tiempo implantados en la industria del software por lo que suelen ser tecnologías muy maduras, mientras que S-Optica es una librería experimental muy limitada. No obstante, ya resuelve muchos de los problemas que están fuertemente arraigados en la aproximación ORM.

En general, las ventajas de S-Optica respecto a los ORMs son muy similares a las que existen entre estos y las aproximaciones funcionales para LINQ, por lo que resulta de interés comparar S-Optica con librerías como Slick o Quill [9]. Esta última está fuertemente inspirada en T-LINQ [19] y sigue los mismos principios teóricos. El principal beneficio de esta librería con respecto a P-LINQ, la implementación original de T-LINQ en F#, es la habilidad de Quill para producir las consultas finales en tiempo de compilación, explotando las facilidades de metaprogramación que ofrece Scala [12]. Desafortunadamente, y a pesar de que Quill proporciona una implementación de `flatMap` para el constructor de tipos `Query` (que actúa como representación para las consultas), carece de una

[6] Siendo más preciso, los valores son objetos en Scala mientras que las consultas de S-Optica son realmente métodos polimórficos. No obstante, se podrían traducir en valores usando una representación basada en Church encoding.

implementación para `point` que complete la interfaz monádica, necesaria para poder ofrecer una traducción de expresiones de S-Optica en expresiones de tipo `Query`. Slick es similar a Quill, pero no se apoya en un lenguaje teórico como T-LINQ[7]. En cualquier caso, ambas librerías mapean los modelos relacionales en Scala de una forma muy directa, es decir, como modelos planos, mientras que S-Optica trabaja con estructuras anidadas y por tanto debe de enfrentarse al problema de la impedancia. Por otro lado, y a pesar de que Quill y Slick soportan actualizaciones y eliminaciones, cubren estos aspectos con lenguajes *ad hoc* que escapan de la interfaz provista por las comprehensions. Optica debería de ser capaz de proporcionar una interfaz estándar para soportar tales facilidades mediante la introducción de nuevas ópticas. Como nota final, se vuelve a destacar que Optica no debería de verse como un competidor de estas librerías, sino más bien como un complemento, ya que se ha demostrado que las ópticas y las comprehensions pueden convivir.

8.5. Optica y Stateless

Esta sección discute las aproximaciones propuestas por Stateless y Optica, con el fin de mostrar cómo Optica resuelve las limitaciones que se enumeraban en la sección 5.3 sobre la aproximación basada en optic algebras. Se utilizará el ejemplo de las parejas para guiar la comparación. La figura 8.1 introduce los tipos de datos y ópticas que serán compartidos por ambas aproximaciones.

```
type SCouples = List[SCouple]
case class SCouple(her: SPerson, him: SPerson)
case class SPerson(name: String, age: Int)

val couplesFl: Fold[SCouples, SCouple] = Fold(_.couples)
val herGt: monocle.Getter[SCouple, SPerson] = Getter(_.her)
val himGt: monocle.Getter[SCouple, SPerson] = Getter(_.him)
val nameGt: monocle.Getter[SPerson, String] = Getter(_.name)
val ageGt: monocle.Getter[SPerson, Int] = Getter(_.age)
```

Figura 8.1: Tipos de datos y ópticas en ejemplo de las parejas.

Modelo de datos La figura 8.2 muestra una posible implementación del modelo de las parejas utilizando Stateless. Como ya se mencionó con anterioridad, este tipo de definiciones no es ni mucho menos trivial: existen demasiados tipos abstractos, se requieren instancias para las algebras anidadas y la propia definición de las optic algebras utilizando `Aux` también resulta bastante confusa. El modelo correspondiente a la librería Optica se muestra en la figura 8.3. En él simplemente es necesario definir las ópticas que seleccionan las partes de interés, que trabajan en torno a las estructuras de datos inmutables determinadas por el dominio. La única complejidad reside en que el tipo de las ópticas debe venir envuelto bajo el tipo para la representación `Repr`, complejidad accidental que surge para dar soporte a las diversas interpretaciones. Por tanto, la definición de capas de datos en Optica resulta muy cercana a la forma de proceder con

[7]Una comparación entre Quill y Slick escrita por el propio desarrollador de Quill puede encontrarse en `https://github.com/getquill/quill/blob/master/SLICK.md`.

ópticas en el plano concreto, a diferencia de la versión de Stateless donde el programador debe lidiar con un ruido innecesario.

```
trait Couples[CS] {
  type P[_]
  type C

  val couple: Couple[C]
  val couples: FoldAlg.Aux[P, couple.P, C]
}

trait Couple[C] {
  type P[_]
  type Pr

  val person: Person[Pr]
  val her: GetterAlg.Aux[P, person.P, Pr]
  val him: GetterAlg.Aux[P, person.P, Pr]
}

trait Person[Pr] {
  type P[_]

  val name: GetterAlg[P, String]
  val age: GetterAlg[P, Int]
}
```

Figura 8.2: Capa de datos del ejemplo de las parejas en Stateless

```
trait Model[Repr[_]] {
  def couples: Repr[Fold[SCouples, SCouple]]
  def her: Repr[Getter[SCouple, SPerson]]
  def him: Repr[Getter[SCouple, SPerson]]
  def name: Repr[Getter[SPerson, String]]
  def age: Repr[Getter[SPerson, Int]]
}
```

Figura 8.3: Capa de datos del ejemplo de las parejas en Optica

Consultas genéricas Una vez definidos los modelos debería de ser posible definir las consultas genéricas. Lamentablemente, no se pueden utilizar `differences` o `expertise` para la comparación, ya que el combinador `fork` no está disponible para la aproximación basada en optic algebras. En este sentido, Optica es más expresivo, ya que la inclusión de cualquier combinador existente en el plano concreto debería ser posible, simplemente haciendo lifting sobre el mismo en el lenguaje. Evidentemente, incluir un nuevo combinador podría restringir el número de interpretaciones soportadas, aunque este aspecto se podría delimitar gracias a la extensibilidad de tagless final. En cualquier caso, la figura 8.4 muestra `under50` (exenta de `fork`s) implementada en términos de Stateless, que consulta el nombre de todas las mujeres menores de 50 años. Como se puede apreciar, se recibe una instancia `cs` de `Couples` y se devuelve un programa de tipo `cs.P`, es decir, un programa que sabe como hacer evolucionar al conjunto de parejas. La signatura también requiere una instancia de `MonadReader`

necesaria para el predicado de `filtered`, combinador que es posible definir en Stateless, aunque no se muestra por brevedad. La figura 8.5 es la implementación de la consulta análoga en Optica, que viene parametrizada con los tipos de representación y observación, y que devuelve precisamente la observación de una consulta. Esto denota la declaratividad de esta versión, donde simplemente se habla de representaciones de consultas y no de programas con efectos como en el caso anterior. Merece la pena destacar que a pesar de esta importante distinción, la implementación es prácticamente idéntica, donde la concisión de la segunda versión se debe principalmente al uso de operadores con nombres más compactos.

```
def under50[CS](
    cs: Couples[CS])(implicit
    MR: MonadReader[cs.couple.person.P, String]): cs.P[List[String]] = {
  import cs.couple._, person._

  cs.couples
    .composeGetter(her)
    .composeFold(filtered(age.get.map(_ < 50)))
    .composeGetter(name)
    .getList
}
```

Figura 8.4: Consulta `under50` en Stateless.

```
def under50[Repr[_], Obs[_]](implicit
    O: Optica[Repr, Obs],
    M: CoupleModel[Repr]): Obs[SCouples => List[String]] =
  (couples ⋙ her ⋙ filtered(age < 50) ⋙ name).getAll
```

Figura 8.5: Consultas `under50` en Optica.

Semántica estándar Si se pretende recuperar el comportamiento habitual de las ópticas para lidiar con estructuras de datos inmutables, se deben proporcionar las evidencias correspondientes. Para el caso de Stateless, es necesario proporcionar una instancia de `Couples` que utilice `Reader` como efecto. La implementación es bastante tediosa y repetitiva, apoyándose en primitivas de Stateless que transforman ópticas en transformaciones naturales (homomorphisms)[8]. La instancia del modelo de Optica, que se muestra en la figura 8.7, simplemente requiere indicar las ópticas en las que se traduce cada primitiva, por lo que su implementación resulta trivial. Si se proporcionan estas intancias a las consultas genéricas correspondientes se generan funciones de tipo `SCouples => List[String]` que son extensionalmente equivalentes en ambos casos.

[8]La instancia de este tipo de algebras para estructuras de datos inmutables se podría automatizar, tal y como se demostró con la implementación de *naturally* [51], una librería que nacía como un complemento a Stateless y que facilitaba el trabajo a la hora de lidiar con transformaciones naturales. No obstante, se desconoce la posibilidad de automatizar instancias para otras infraestructuras más complejas.

```
val personR = new Person[SPerson] {
  type P[x] = Reader[SPerson, x]

  val name = fromGetter(nameGt)
  val age = fromGetter(ageGt)
}

val coupleR = new Couple[SCouple] {
  type P[x] = Reader[SCouple, x]
  type Pr = SPerson

  val person = personR
  val her = fromGetter(herGt)
  val him = fromGetter(himGt)
}

implicit val couplesR = new Couples[SCouples] {
  type P[x] = Reader[SCouples, x]
  type C = SCouple

  val couple = coupleR
  val couples = fromFold(couplesFl)
}
```

Figura 8.6: Instancia de modelo para ejemplo de parejas en Stateless.

```
implicit object RModel extends Model[λ[x => x]] {
  val couples = couplesFl
  val her = herGt
  val him = himGt
  val name = nameGt
  val age = ageGt
}
```

Figura 8.7: Instancia de modelo para ejemplo de parejas en Optica.

Semánticas no estándar A pesar de que el resultado de la aproximación basada en optic algebras consigue un resultado final muy cercano al de Optica en términos de eficiencia, no ocurre lo mismo en otras interpretaciones. Uno de los principales problemas de Stateless es que la implementación de los combinadores estándar viene prefijada. Por ejemplo, la composición de optic algebras se corresponde con la composición de transformaciones naturales. Sin embargo, Optica otorga total libertad al programador de los intérpretes para que lleve a cabo la evaluación que considere más conveniente, lo que supone una enorme flexibilidad. El otro gran problema reside en que los programas generados en Stateless son efectos en lugar de representaciones. Consecuentemente, resulta muy complejo llevar a cabo optimizaciones sobre ellos, ya que la introspección de los programas es compleja o simplemente inviable. En el caso de Optica, los programas se corresponden con representaciones más generales, tal y como vimos con las tripletas, que permiten introspección y ofrecen la posibilidad de introducir fuertes optimizaciones. En este caso particular, se garantiza la creación de una única consulta SQL final, mientras que la aproximación ofrecida por Stateless en su estado actual requiere un acceso a la base de datos adicional por

cada nivel de anidamiento.

Consideraciones finales Una de las limitaciones asociadas a la aproximación de Stateless que no se ha contemplado en los párrafos anteriores es que, a excepción de lens algebra, las optic algebras no están formalizadas. En cierta manera, este aspecto refleja los problemas de extensibilidad existentes en Stateless, donde la introducción de nuevas ópticas requiere un tedioso proceso de formalización en búsqueda de abstracciones análogas en el plano genérico. En cambio, en el caso de Optica, este aspecto simplemente requiere la inclusión de nuevos tipos de óptica y primitivas en el lenguaje, aunque este aspecto afectaría a los intérpretes. Por otro lado, se podría intuir que Stateless sí que tiene una ventaja frente a Optica, y es su soporte para actualizaciones. Sin embargo, las actualizaciones resultan ser *ad hoc* para ejemplos específicos por lo que no se ofrecen ningunas garantías, ya que la investigación en esta línea se abandonó al ver los resultados tan prometedores de Optica, donde abordar las actualizaciones de forma eficiente se contempla como trabajo futuro.

Parte IV

Conclusiones

La última parte de este documento destaca los aspectos más relevantes de la investigación desarrollada en las partes II y III, apuntando hacia las nuevas líneas de investigación que se abren a partir de los resultados obtenidos (capítulo 9).

Capítulo 9

Conclusiones y trabajo futuro

Finalmente se llega al último capítulo de esta tesis doctoral, donde se concluyen los resultados obtenidos en la parte II y los correspondientes a la parte III. Estas partes desarrollaban los objetivos (ver sección 1.4) con los que se pretendía abordar la hipótesis de trabajo, que se vuelve a formular aquí por completitud:

Hipótesis *Es posible diseñar lenguajes de consultas que se inspiren en las abstracciones y patrones de composición propios de las ópticas, permitiendo ampliar el rango de actuación de éstas más allá de las estructuras de datos inmutables en memoria. Los lenguajes resultantes deberían de mitigar los problemas asociados a los repositorios funcionales y a las soluciones LINQ basadas en comprehensions descritos previamente.*

En particular, la sección 9.1 se postula como una respuesta a los objetivos 1 y 2, mientras que la sección 9.2 hace lo propio para los objetivos 3 y 4, con lo que se tratará de demostrar la viabilidad de la hipótesis. Antes de dar fin al capítulo, se apuntará hacia trabajo futuro (sección 9.3) no sólo a un nivel académico sino también desde una perspectiva más industrial.

9.1. Optic algebras y Stateless

La línea de investigación seguida a lo largo de la parte II trataba de encontrar analogías entre las ópticas y las teorías algebraicas recogidas en MTL, con el objetivo de trasladar los beneficios de las ópticas a un plano genérico de computaciones abstractas. El trabajo quedaba organizado en torno a dos objetivos (sección 1.4). El capítulo 3 desarrolla y da respuesta al **Obj. 1** de esta tesis mediante las contribuciones que se reflejan en los siguiente puntos:

- Se formaliza cuál es la conexión precisa que existe entre *MonadState* y una *very well-behaved lens*: la primera abstracción es una generalización de la segunda. De hecho, esta conexión ha favorecido el descubrimiento de otra nueva representación para lens (bajo la forma de una instancia particular de `MonadState`) que bien podría incluirse en la lista recogida en la

sección 2.1.2. A pesar de que esta representación no ofrece a priori ningún beneficio para su explotación en una librería convencional de ópticas, sí que arroja una nueva perspectiva sobre `MonadState` que resulta de gran interés para este trabajo: esta teoría algebraica puede entenderse como el análogo a `lens` en el plano genérico. Con el objetivo de resaltar esta conexión, se propone *lens algebra* como alias para dicha abstracción. La nueva intuición adquirida permite utilizar este concepto como un agregador estándar de las primitivas de lectura y manipulación de estado que suelen encontrarse en los repositorios funcionales, con lo que se puede reducir su tamaño notablemente.

- El potencial de una `lens` se alcanza en su totalidad gracias a sus capacidades de composición, por lo que este aspecto también debía de llevarse al plano genérico. Para esta tarea, la investigación se apoya en la representación de óptica que se muestra en la definición 14, sobre la que se lleva a cabo un proceso de abstracción que permite el descubrimiento de una representación alternativa de `MonadState` en términos de un morfismo de mónadas. La falta de heterogeneidad empuja a llevar el proceso de abstracción un paso más allá, resultando en el concepto de *lens algebra homomorphism*, con el que se habilita la composición de las lens algebras en el plano genérico. La principal ganancia obtenida desde una perspectiva pragmática es la posibilidad de definir repositorios funcionales anidados. No obstante, su definición resulta demasiado compleja para su aplicación industrial.

- Las pautas seguidas para el surgimiento de un lens algebra se generalizan y se propone un proceso de diseño para descubrir las teorías algebraicas asociadas a otras ópticas de la jerarquía. En este sentido se identifican varios retos que van a tener que abordarse para posibilitar el proceso de diseño de otras optic algebras: se requiere el diseño de nuevas teorías algebraicas asociadas a otras ópticas y es necesario dar con la representación basada en morfismos de mónadas para éstas. Este proceso se aplica experimentalmente en Stateless, como se muestra a continuación.

El capítulo 4 cubría el **Obj. 2** de la tesis, mediante las contribuciones que se listan a continuación:

- Se propone Stateless como una librería escrita en Scala que implementa las ideas del capítulo 3 y que aplica el proceso de diseño para descubrir nuevas abstracciones. La librería empaqueta un rico catálogo de optic algebras y una noción de composición que se apoya en una representación experimental de las ópticas en términos de morfismos de mónadas. La composición heterogénea de estas abstracciones satisface unas reglas análogas a las que se establecen en la jerarquía de las ópticas (figura 1.1). Sin embargo, las nuevas abstracciones son completamente experimentales, ya que se apoyan en representaciones de ópticas no formalizadas.

- La validación de la librería con varios ejemplos sirve para identificar algunas abstracciones que podrían ser comunes a muchas aplicaciones, destacando la adaptación de `At` y `FilterIndex`, y la abstracción recogida por `MapAlg` para encapsular el comportamiento de un mapa en el plano general. En particular, éste último se apoya en las ópticas indexadas, otra

abstracción experimental recogida en Stateless. Gracias a la introducción de estos patrones se logra que la implementación de la lógica de negocio sea elegante, aunque se aprecia una falta de expresividad, ya que no todos los combinadores existentes en el plano concreto pueden llevarse al nuevo escenario.

- Las abstracciones de Stateless ofrecen dos interpretaciones, con el objetivo de mostrar la generalidad de las consultas. En primer lugar, se recupera el comportamiento estándar de las ópticas, que se apoya en la librería Monocle [119]. En segundo lugar, se ofrece una instanciación para bases de datos relacionales, donde se utiliza Doobie [94] como intermediario. Por tanto, se logra la generalidad esperada, pero las interpretaciones resultan ser muy ineficientes, con poco margen de optimización.

La aproximación de Stateless basada en optic algebras, a pesar de cumplir con los objetivos estipulados, deja una sensación agridulce: todas las limitaciones encontradas hacen que su aplicación industrial no sea viable. Por suerte, todas ellas desaparecen en la línea de investigación paralela, tal y como se muestra en la próxima sección.

9.2. Optica y S-Optica

La línea de investigación alternativa descrita en la parte III de esta tesis pretendía seguir una aproximación más directa, embebiendo un subconjunto de ópticas y algunas de sus primitivas estándar más habituales en un nuevo DSL, postulándose así como una alternativa a las comprehensions para hacer LINQ. El capítulo 6 abordaba el **Obj. 3** que se logra gracias a los siguientes hitos:

- Se propone Optica, un lenguaje que pretende especificar qué es lo que las diferentes representaciones de ópticas tienen en común. En particular, se trata de un lenguaje que se nutre de un subconjunto de ópticas (de *sólo lectura*) y algunos de sus combinadores habituales para definir su sintaxis y sistema de tipos. Se establece el mecanismo para extender el core del lenguaje con las primitivas específicas de cada dominio. Finalmente, se muestra cómo componer consultas genéricas con una notación *point-free* muy modular, que más tarde serán traducidas a consultas específicas para diversas tecnologías de manera eficiente. La principal ventaja de esta aproximación es que no requiere una compleja búsqueda de análogos en el mundo de las computaciones: cualquier óptica o combinador son candidatos potenciales para ser incluidos en el lenguaje.

- Se identifican las ópticas concretas como la semántica estándar del lenguaje, con la que se puede recuperar el comportamiento convencional de acceso y manipulación de estructuras de datos inmutables. Adicionalmente, se ofrecen traducciones no estándar para XQuery y SQL. La primera de ellas resulta ser muy elegante, lo que confirma la idoneidad del lenguaje para lidiar con infraestructuras basadas en modelos anidados. La interpretación a SQL no es ni mucho menos trivial, donde se utiliza el concepto de `Triplet` como semántica no estándar, que posibilita la aplicación de optimizaciones agresivas. El aspecto más relevante que deriva de estas interpretaciones es que permiten entender estos dominios semánticos como

representaciones alternativas de ópticas: su propósito principal también es el de seleccionar 'partes' contextualizadas en un 'todo'. No obstante, estas representaciones no se comprometen con las estructuras de dato en memoria y habilitan la posibilidad de trabajar con fuentes de datos alternativas, pero se benefician de la misma elegancia y declaratividad que manifiestan sus análogas en el plano concreto.

- Se establece que el lenguaje de ópticas, aun siendo una alternativa interesante a las comprehensions para modelos anidados, también puede interoperar con éstas. Esta idea se formaliza con una interpretación del lenguaje a T-LINQ [19], que evidencia la interoperabilidad entre ambas aproximaciones a LINQ. De esta manera, las librerías actuales basadas en comprehensions, como Slick o Quill en el ecosistemas de Scala, podrían beneficiarse de la introducción de ópticas como primitivas del lenguaje, especialmente para lidiar con modelos anidados. Merece la pena destacar como limitación que no todo modelo relacional es expresable mediante ópticas, impedimento que también se muestra en Optica.

El **Obj. 4** se completa en el capítulo 7 mediante las siguientes contribuciones:

- Se propone S-Optica [52], librería que embebe Optica en el lenguaje de programación Scala. Se utiliza la aproximación tagless-final [13] para llevar a cabo la implementación del lenguaje. La implementación en sí misma es una contribución interesante, ya que ilustra la aplicación de estas técnicas de embedding en el caso particular de Scala. La adopción de tagless-final permite que la librería sea fácilmente extensible, lo cual resulta muy conveniente ante la contemplación de nuevas ópticas y combinadores. Es importante destacar también que la definición de los modelos de las aplicaciones y la implementación de la lógica de negocio tiene una complejidad accidental muy reducida, especialmente si se compara con Stateless.

- Se implementan las interpretaciones estándar y no estándar (XQuery, SQL y T-LINQ) y se comprueba que la traducción de las consultas genéricas produce consultas específicas correctas, tal y como se muestra en los tests desplegados a lo largo de la librería. El estilo más general de tagless-final, donde los dominios semánticos no necesariamente se corresponden con computaciones (como sí ocurría en MTL) se convierte en una pieza clave para introducir optimizaciones y ofrecer unas garantías mínimas de eficiencia en las traducciones.

Optica permite por tanto llevar las abstracciones y los patrones de composición de las ópticas a un plano genérico, cuyo rango de actuación incluye, pero no esta restringido a estructuras de datos inmutables. Como bonus adicional, las consultas genéricas permiten optimizaciones muy agresivas durante las traducciones a las consultas específicas, garantizando unas condiciones mínimas de calidad. Además se evitan los problemas asociados a los repositorios funcionales: la definición de modelos anidados es viable y sencilla; y a las soluciones LINQ basadas en comprehensions: se ofrece una alternativa point-free para modelos anidados donde el lenguaje de actualizaciones es estándar.

9.3. Trabajo futuro

El trabajo de investigación presentado en esta tesis abre nuevas líneas de investigación académicas y favorece el surgimiento de nuevas posibilidades en el ámbito industrial. Esta sección recoge las líneas de actuación más interesantes que se podrían acometer tras la consecución de esta tesis.

Plano académico

- Uno de los problemas abiertos de LINQ es la adaptación a otros modelos de programación centrados en el dato [16]. La elegancia de la interpretación de consultas de Optica a XQuery permite inferir los beneficios de la aproximación para lidiar con modelos anidados. Partiendo de esta motivación, y con el fin de probar la generalidad de Optica, se pretende abordar la implementación de interpretaciones eficientes e idiomáticas a tecnologías tales como bases de datos NoSQL orientadas a documentos [53], lenguajes de consultas web como GraphQL [54] y en general tecnologías *warehouse* destinadas a la analítica de datos.

- La interpretación a SQL de Optica evidencia una expresividad más limitadas que la que ofrece una aproximación LINQ basada en comprehensions, ya que los joins no están soportados. Se pretende investigar en posibles extensiones de Optica basadas en la codificación composicional de equijoins en [38]. También se pretende investigar en nuevas interpretaciones de Optica en lenguajes de consultas declarativos como Datalog [14] y lógicas de descripción [77], así como establecer conexiones con desarrollos más recientes relacionados con lenguajes de comprehensions basados en monoides [29].

- La extensión del lenguaje con nuevas ópticas (lenses, affine traversals, traversals, etc.) y la inclusión de combinadores adicionales que suelen encontrarse las librerías estándar de ópticas sería el siguiente paso natural. Este aspecto requeriría prestar especial atención a las leyes que las posibles representaciones, ya sean estándar o no estándar, deben satisfacer. Recordamos que hasta ahora se ha podido ignorar este aspecto, ya que las ópticas de sólo lectura carecen de leyes. Tal y como apuntaba uno de los revisores anónimos de [87], podría resultar muy interesante analizar la forma final que adquieren estas leyes en las diversas interpretaciones, ya que podrían llegar a identificarse nuevas optimizaciones sobre éstas. Finalmente, merece la pena destacar que la extensión de S-Optica con nuevas abstracciones se podrá beneficiar del embedding tagless-final.

- Las ópticas del punto anterior mostrarían un gran potencial para lidiar no sólo con consultas de lectura sino también con actualizaciones, un aspecto que se ha descuidado en el campo de LINQ, tal y como se argumenta en [16]. Este trabajo de investigación consolida los pilares para abordarlas como trabajo futuro. La posibilidad de introducir actualizaciones en los intérpretes está sujeta a las limitaciones impuestas por la propia infraestructura. Por ejemplo, XQuery no soporta actualizaciones (aunque existen extensiones que abordan este aspecto [7]), y consecuentemente la evaluación de ópticas de escritura sería parcial. Para el caso de SQL, sí que

se soportan actualizaciones, pero se compensa con falta de expresividad: no todas las consultas relacionales se pueden tratar como actualizaciones, lo que introduce un nuevo nivel de parcialidad. El hecho de que las tripletas necesiten ser extendidas para acomodar actualizaciones es algo que requiere más investigación. En general, existe un gran optimismo por el potencial de las actualizaciones para lidiar con tecnologías modernas basadas en modelos anidados como las descritas en los párrafos anteriores. Finalmente, es importante destacar que la versión polimórfica de las ópticas podría entenderse como una actualización de los esquemas subyacentes. Por ejemplo, una lens que transforme una selección textual en numérica, podría interpretarse en SQL como una sentencia `ALTER` que alterase el tipo de la columnna correspondiente en la tabla involucrada.

- Se ha implementado una prueba de concepto de las ideas de Optica en la librería S-Optica usando el estilo tagless-final para embeber el lenguaje. Optica se implementa por tanto mediante una type class: la clase de representaciones de ópticas y sus consultas asociadas. Además de las consultas mostradas para guiar las explicaciones, se ha probado S-Optica con múltiples consultas sobre estos dominios y sobre otros nuevos obtenidos de la documentación oficial de Monocle, Slick y Quill. Estos ejemplos se encuentran localizados en una rama experimental de S-Optica y serán liberados cuando la librería alcance un mayor nivel de madurez. En este sentido, se pretende aprovechar las novedades de metaprogramación y *multi-stage programming* [108] que vendrán incluidas en *Dotty* [99], la nueva versión de Scala —que todavía no ha sido publicada a fecha de hoy— para simplificar las implementaciones. Sin necesidad de centrarnos en las ópticas, la investigación llevada a cabo deja la sensación de que la enorme proliferación de tecnologías que ha tenido lugar en los últimos años se podría beneficiar del diseño de DSLs que posibiliten traducciones idiomáticas y eficientes, bajo el eslogan de *abstracción sin culpa*.

Plano industrial

- A pesar de que el trabajo en Stateless ha quedado en la sombra, el conocimiento adquirido durante la elaboración de las pruebas formales con el asistente de pruebas Coq ha sido muy fructífero. En este sentido, se ha manifestado un fuerte interés por la programación basada en tipos dependientes, también con otros lenguajes como Idris [8] o Agda, que abren las puertas a futuros proyectos orientados en torno a la programación certificada.

- La línea industrial más evidente es probablemente la de pulir y extender Optica para que ofrezca unas garantías de calidad y usabilidad mínimas. Esto requeriría abordar algunos de los puntos del plano académico (nuevos combinadores, ópticas e intérpretes) y hacer mucho trabajo de difusión en forma de blogs, charlas en meetups locales y ponencias en conferencias internacionales.

- Se podrían llevar a cabo implementaciones de Optica a otros lenguajes de programación. En este sentido, destacamos una posible colaboración con la empresa *47 degrees* para migrar las ideas recogidas en esta tesis

a la librería *Arrow* [3] escrita en Kotlin, lenguaje que ha adquirido gran notoriedad en la industria del software en estos últimos años.

- Los resultados obtenidos en la sección 6.5 son suficientemente alentadores como para anticipar la posibilidad de hacer contribuciones en las librerías de LINQ basadas en comprehensions existentes en el ecosistema de Scala, como Quill [9] o Slick, a fin de extenderlas con soporte para ópticas.

- Otra línea de actuación pasaría por la generalización de las librerías convencionales de ópticas, mediante la introducción de un DSL para recoger las abstracciones y primitivas de las ópticas, y asumiendo que su semántica estándar se corresponde con el comportamiento actual de la librería. Esta tarea tendría un impacto acusado sobre el código existente, por lo que su puesta en funcionamiento no sería trivial. Afortunadamente, Habla Computing se ha posicionado como una empresa experta en ópticas, por lo que sus miembros ya están tomando parte activa en las discusiones para la adaptación de Monocle a Dotty [120].

- El trabajo relacionado en esta investigación ha permitido el estudio de los fundamentos del *multi-stage programming*. Este aspecto será una pieza central en Dotty, hecho que coloca a Habla Computing en una posición privilegiada para la captación de nuevos clientes interesados en DSLs. Merece la pena destacar que ya se han empezado a aplicar las técnicas aprendidas en esta investigación sobre tagless-final en un proyecto del sector energético, obteniendo muy buenos resultados.

- Se pretende utilizar los conocimientos adquiridos para la implementación de diversas librerías propias destinadas a la comunidad de programadores de Scala. En este sentido, ya se han identificado varias ideas en el campo de la analítica de datos. El objetivo principal es que Habla Computing se afiance como experta internacional en el paradigma de programación funcional y en el desarrollo de DSLs, aunque esta frase resulta ciertamente redundante, ya que aludiendo a Wadler: "A functional language is a domain-specific language for creating domain-specific languages".

La realización de esta tesis doctoral ha posibilitado la exploración del terreno pantanoso que existe entre la academia y la industria, que en muchas ocasiones parecen avanzar por caminos completamente independientes. Desde nuestra perspectiva, creemos firmemente que una parte importante de la investigación debe nutrirse de problemas de la industria y, de la misma manera, una parte importante de los problemas de la industria deben apoyarse en soluciones propuestas por la academia. La búsqueda de un modelo de negocio para Habla Computing que permita la puesta en funcionamiento y el mantenimiento de este proceso cíclico de forma estable e indefinida es uno de los principales retos al que nos enfrentamos.

Apéndice A

Conceptos básicos y patrones de Scala

Esta sección introduce muy brevemente los conceptos y patrones de Scala que se utilizan en este trabajo de investigación. En primer lugar, la tabla A.1 proporciona ejemplos y breves descripciones de las abstracciones y definiciones que se han considerado más relevantes. Como se puede apreciar, algunas de ellas son exclusivas de Scala, mientras que otras están muy extendidas en la comunidad de programación funcional y simplemente mostramos su adaptación en este lenguaje. En segundo lugar, se describe el patrón general que se suele seguir en Scala [101] para codificar una type class [129].

A.1. Codificando type classes en Scala

En Scala, es posible definir nuevas type classes utilizando *traits*. Por ejemplo, podemos codificar *Functor*, una clase inspirada en Teoría de Categorías muy extendida en la comunidad de programación funcional, de la siguiente manera:

```
trait Functor[F[_]] {
  def fmap[A, B](fa: F[A])(f: A => B): F[B]
}
```

El trait está parametrizado con el constructor de tipos `F[_]`; por lo tanto esta clase es en realidad una type constructor class, donde `F` representa a la clase de tipos que son (endo)funtores. Esta clase declara un único método como parte de su interfaz, `fmap`, que está parametrizado por los tipos `A` y `B` y por los valores `fa` y `f`. Como se puede apreciar, `fa` y `f` no vienen definidos en el mismo bloque de parámetros. Este mecanismo es provisto por Scala para introducir una noción de *currificación*. En este caso particular, la separación resulta ser de gran utilidad para inferir el tipo `A` del segundo argumento, cuando se lleve a cabo una invocación de `fmap`. Se podría seguir el mismo proceso para definir nuevas type classes, por ejemplo `Pointed`:

```
trait Pointed[F[_]] {
  def point[A](a: A): F[A]
}
```

o incluso `Bind`:

Abstraction	Code Example	Description

algebraic data type	`sealed abstract class Option[A]`
	`case class None[A]() extends Option[A]`
	`case class Some[A](a: A) extends Option[A]`
		Los ADTs se implementan usando herencia *sellada* de objetos. El ejemplo muestra `Option`, habitualmente conocido como *Maybe*.

case class	`case class Person(`
	`  name: String,`
	`  age: Int)`
		Introduce clases con características especiales, como facilidades de construcción y observación.

companion object	`trait Person` `object Person`	Módulo que acompaña a una clase o trait y que tiene su mismo nombre. Se usa para definir miembros de la clase y definiciones implícitas, como conversores e instancias de type classes.

for comprehension	`for {`
	`  i ← List(1, 2)`
	`  j ← List(3, 4)`
	`} yield i + j`
	`// res: List[Int] = List(4,5,5,6)`
		Azúcar sintáctico para `flatMap`, `map`, etc. Análogo a la *do notation* en Haskell.

function type	`val f: Int => Boolean = i => i > 0` `f(3) // res: Boolean = true`	Los tipos función se representan con flechas que separan el dominio del codominio. Las expresiones lambda siguen una notación similar, donde la flecha separa los parámetros del cuerpo de la función.

implicit resolution	`def isum(x: Int)(implicit y: Int): Int =`
	`  x + y`
	
	`implicit val i: Int = 3`
	`isum(1) // res: Int = 4`
		Familia de técnicas que permiten al compilador inferir determinados argumentos automáticamente. En el ejemplo, `i` es pasado como segundo argumento a `isum` implícitamente.

partial function syntax	`val f: Option[Int] => Boolean = {`
	`  case None => true`
	`  case Some(_) => false`
	`}`
		Sintaxis especial para las situaciones en las que se quiere producir una función (potencialmente parcial) que requiere pattern matching sobre su parámetro.

pattern matching	`opt match {`
	`  case None => true`
	`  case Some(_) => false`
	`}`
		Busca patrones en un valor y ejecuta el código asociado al primero que encaje, si existe.

placeholder syntax	`val inc:  Int => Int = i => i + 1` `val inc2: Int => Int = _ + 1`	Sintaxis para expresiones lambda donde nos referimos al parámetro como _. Por lo tanto, no hay necesidad de ponerle un nombre. Tanto `inc` como `inc2` (versión placeholder) son equivalentes.

trait	`trait Person {`
	`  def name: String = "John"`
	`  def age: Int`
	`}`
		Similar a las interfaces de Java (permiten herencia múltiple), pero soportan implementación parcial de sus miembros.

type lambda	`λ[x => x]`
	`λ[x => Int]`
	`λ[x => Option[x]]`
		Notación proporcionada por el plugin del compilador *kind-projector* para producir funciones de tipo anónimas con kind $* \to *$.

type member	`trait A {`
	`  type B`
	`}`
		Tipo definido en el contexto de un trait o clase. En el ejemplo, `B` se corresponde con un type member abstracto.

type parameter	`trait List[A]`
	`def nil[A]: List[A]`
	`trait Symantics[Repr[_]]`
		Tipos que se toman como parámetros por clases o métodos. Se requiere una notación especial si se esperan tipos de higher kind, como `Repr`.

Tabla A.1: Conceptos básicos del lenguaje Scala.

```
trait Bind[F[_]] {
  def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

donde se sigue exactamente el mismo patrón: se define un trait parametrizado por un tipo y se definen los métodos abstractos que lo caracterizan. Scala es un lenguaje que combina la programación funcional con la orientación a objetos. Como resultado, las clases pueden componerse de una forma peculiar, mediante herencia. De hecho, es posible definir `Monad` en términos de todas las type classes que se han introducido con anterioridad, tal y como se muestra a continuación:

```
trait Monad[F[_]] extends Functor[F] with Pointed[F] with Bind[F] {
  def fmap[A, B](fa: F[A])(f: A => B) = bind(fa)(a => point(f(a)))
}
```

Esta definición explota los mecanismos de herencia múltiple provistos por Scala para combinar todos los traits. La nueva abstracción ofrece una implementación de `fmap` que se deriva a partir de `point` y `bind`. Merece la pena destacar que `Monad` sigue siendo abstracta, ya que todavía no se ha proporcionado una implementación para tales primitivas.

Desplegar las intancias de las type classes en los companion object de la type class a la que acompañan resulta ser una práctica muy habitual, ya que el compilador de Scala las busca aquí por defecto, entre otras localizaciones. De esta manera, se podría definir un companion object para `Monad` en el que se incluya una instancia de dicha type class para el tipo de datos `Option`, como se muestra en la figura A.1.

```
object Monad {
  implicit object OptionMonad extends Monad[Option] {
    def point[A](a: A) = Some(a)
    def bind[A, B](fa: Option[A])(f: A => Option[B]) = fa match {
      case None => None
      case Some(a) => f(a)
    }
  }
}
```

Figura A.1: Instancia de mónada `Option`.

Existen varias alternativas para introducir esta instancia: como un valor, como una definición, etc. En este caso particular se ha utilizado un objeto implícito de nombre `OptionMonad`, que queda a disposición del compilador para cuando crea conveniente utilizarlo. La implementación de `point` y `bind` resulta trivial, requiriendo pattern matching para el segundo caso. Uno de los aspectos más fundamentales a la hora de definir una type class es su expresividad, es decir, los combinadores primitivos deberían abrir las puertas a un amplio catálogo de operaciones derivadas. Por ejemplo, las primitivas de `Monad` nos permiten implementar el método `join`, más cercano a la definición matemática de una mónada:

```
def join[F[_], A](ffa: F[F[A]])(implicit M: Monad[F]): F[A] =
  M.bind(mma)(identity)
```

Este método requiere una evidencia implícita de `Monad` (`M`) para `F`, es decir, se requiere que `F` sea una instancia de `Monad`. Dicha evidencia es utilizada por la

implementación para invocar al método `bind`. En Scala suelen existir numerosos caminos para implementar la misma funcionalidad. De hecho, introduciendo la notación de context bound y proporcionando ayudas sintácticas (que no mostramos por concisión) podríamos llegar a la siguiente definición para `join`:

```
def join[F[_]: Monad, A](ffa: F[F[A]]): F[A] =
  mma ≫= identity
```

que el programador de Haskell encontrará más natural.

La definición de `join` debería de permitir el aplanamiento de valores opcionales anidados, tal y como manifiesta este código:

```
join[Option, Int](Option(Option(3)))(Monad.OptionMonad)
// res: Option[Int] = Some(3)
```

En esta situación, todos los argumentos (tipos incluidos) se proporcionan de forma explícita. Por suerte los mecanismos de inferencia de Scala habilitan la siguiente versión, mucho más sencilla, pero con el mismo comportamiento:

```
join(Option(Option(3)))
// res: Option[Int] = Some(3)
```

Como nota final, merece la pena destacar que una instancia de `Monad` también es una instancia válida para `Functor`, `Pointed` y `Bind`. Esto puede resultar chocante para un programador de Haskell, donde las clases vienen definidas con constraints que fuerzan al programador a proporcionar instancias para las dependencias antes de llevar a cabo la implementación de la instancia que las combina.

Apéndice B

Contexto de Descubrimiento

Esta tesis se desarrolla en un ambiente industrial y parte motivada por un problema quizás demasiado amplio. La larga búsqueda de un foco más concreto y posiblemente la ausencia de una comunicación inicial con otros investigadores del sector derivaron en acusadas reorientaciones en la investigación, callejones sin salida, líneas que se abandonaron por otras más prometedoras, etc. Por todo ello, hemos creído interesante escribir esta sección, donde se muestra una perspectiva alternativa sobre el desarrollo de esta tesis, en la que se describen en un tono más informal los principales hitos, impedimentos e ideas descartadas, siguiendo un orden temporal.

Antecedentes

En la actualidad, Habla Computing centra su negocio en torno al diseño de arquitecturas software basadas en el paradigma funcional (consultoría, formación, auditorías, etc.). Sin embargo, esta pequeña empresa surgía en 2011 para la transferencia tecnológica del lenguaje de programación Speech [109, 110]. Tras un estudio inicial de las alternativas existentes para llevar a cabo la implementación, se decide embeber el lenguaje en Scala, por ser una aproximación que estaba adquiriendo cierta notoriedad por aquel entonces [35]. No fue mucho después cuando empezamos a darnos cuenta de que la combinación del paradigma de orientación a objetos con el paradigma de programación funcional exhibía muchas limitaciones que no se mostraban en los libros, aspecto que acusamos especialmente en la combinación de herencia con inmutabilidad. Para aliviar esta impedancia decidimos desarrollar *updatable* [46], que explotaba las recién llegadas técnicas de metaprogramación de Scala [12] para facilitar la actualización de objetos. Cubrir este aspecto resultaba esencial para permitir la evolución de las diversas entidades existentes en una aplicación de Speech. A pesar de que la librería llegó a ponerse en producción en una importante empresa de banca, tenía una implementación excesivamente compleja[1], acusaba unos tiempos de

[1] `https://github.com/hablapps/updatable/blob/8b531fb370afc9a50b92d79dc8798a7806b66173/src/main/scala/org/hablapps/updatable/metaModel.scala`

compilación muy elevados y requería la carga de los datos en memoria para ejecutar las actualizaciones, lo que derivaba en una aproximación poco eficiente. Encontrar una arquitectura elegante y eficiente para el diseño de la capa reactiva de Speech se convirtió en la motivación principal de esta tesis doctoral, que comenzaría varios años después.

Focalizando el problema

La primera etapa de la tesis estuvo marcada por un aprendizaje más profundo de las técnicas y abstracciones propias del paradigma de programación funcional [81, 21]. Esta tarea resultaba fundamental para poder abordar la literatura de interés, que en un primer momento giraba en torno a las técnicas de *funcional reactive programming* [59], donde el concepto de *behavior* parecía quedar cerca de nuestra noción de entidad. No obstante, esta línea de investigación quedó abandonada cuando nos topamos con la noción de *codata* [121], que contempla estructuras de datos potencialmente infinitas, y que permite la formalización de los sistemas de información en términos de máquinas.

En particular, el concepto de *coalgebra* [65] permite fundamentar el desarrollo de sistemas reactivos mediante *teoría de categorías* [4]. Nuestro interés por esta idea queda reflejado en *coalgebraz* [47], una librería escrita en Scala que persigue la exploración del diseño modular de sistemas, utilizando las coalgebras (o máquinas) como elemento central. La librería incluye máquinas estándar tales como streams, autómatas[2], o incluso objetos [64]. Adicionalmente, la librería incluye nuestro propio tipo de coalgebra, bautizada como *entity*[3], compuesta a partir de "piezas" de otras máquinas conocidas, resultando en una máquina capaz de recibir entrada, volcar salida, observar el estado actual y finalizar su ejecución.

Utilizando este último tipo de máquina como bloque fundamental, se aborda la implementación de varias aplicaciones para validar la librería; concretamente, se lleva a cabo una versión simplificada del popular juego Candy Crush y una aplicación de *Internet of Things* (IoT) basada en *geocercas*. En estas aplicaciones se pone de manifiesto el alto grado de modularidad alcanzado, uno de los principales objetivos perseguidos. No obstante, no se alcanza un correcto desacoplamiento entre la lógica de las aplicaciones y su aterrizaje en una infraestructura, con lo que esta librería no supera los estándares de calidad requeridos para su aplicabilidad en el sector industrial. A pesar de esta limitación, los experimentos realizados dejaban la sensación de ir por buen camino. En este sentido, destacamos el hecho de que una coalgebra puede entenderse como un álgebra con una interpretación basada en estado[83], lo que nos abre la posibilidad de plantear los resultados obtenidos en coalgebraz bajo una nueva perspectiva más algebraica.

Descubrimiento de las *Optic Algebras*

Las álgebras de ópticas surgen tras la destilación de la esencia algebraica que se esconde tras una óptica, tal y como se recoge en [82]. La extracción de

[2] http://blog.sigfpe.com/2006/03/coalgebras-and-automata.html

[3] https://github.com/hablapps/coalgebraz/blob/master/core/src/main/scala/coalgebraz/Entity.scala

este algebra abría finalmente la posibilidad de llevar la evolución de un estado a otras infraestructuras, tales como bases de datos o microservicios. Las álgebras de ópticas, tal y como se presentaban originariamente, no exhiben mecanismos de composición análogos a los que pueden encontrarse en las ópticas. Por ello, inspirados en las representaciones de ópticas van Laarhoven y profunctor propusimos una representación alternativa para nuestras álgebras en forma de transformaciones naturales, que más tarde —y gracias a los trabajos de Abou-Saleh *et al.* [2, 1]— descubriríamos que se fundamentaban en la definición 14 propuesta por Shkaravska [111].

Todas estas ideas, junto con las otras abstracciones experimentales descritas en el capítulo 4 fueron recogidas en *stateless* [48]. En su liberación, el repositorio contenía más de 5000 líneas de código para representar las versiones *cruda* y *natural* de las optic algebras y contaba con interpretaciones a Monocle [119] y Doobie [94] como infraestructuras subyacentes. Se había conseguido generalizar la capa de estado portando la esencia de las ópticas a un plano genérico, lo que resultaba ser un hito muy importante. Desafortunadamente, nos dimos cuenta de que todavía existía un gran impedimento entre gran parte de la comunidad de programación funcional, más acusado incluso en la comunidad de Scala: las ópticas eran todavía unas grandes desconocidas.

Bajo este panorama y para aportar nuestro granito de arena, nos propusimos dar difusión a este tema mediante tutoriales, talleres y charlas en grupos locales (*Scala Programming @ Madrid*, *Madrid Haskell Users Group* y *Functional Programming Madrid*) y en conferencias internacionales (*Lambda World 2017*). La buena aceptación de éstos, destacando la gran acogida de la serie de artículos [84] nos posicionó como expertos en un tema que poco a poco había ido adquiriendo mayor notoriedad. Esto nos permitió realizar varias contribuciones a Monocle[4] y contar nuestro trabajo sobre optic algebras en *Scala eXchange 2017*, una de las principales conferencias sobre Scala a nivel mundial, donde pudimos presentar stateless a la comunidad. El feedback recibido nos confirmaba que la idea era buena, pero que la librería estaba lejos de su explotación comercial: las abstracciones no eran adecuadas[5] y la aproximación, que no daba mucho margen para introducir optimizaciones, resultaba ser muy ineficiente.

De manera paralela, se trabajó en la fundamentación de las abstracciones que se habían identificado en la librería, aspecto esencial para poder presentarlas en un marco académico. En el caso particular de las optic algebras, resultaba inevitable realizar pruebas formales. La complejidad y longitud de las mismas hizo evidente la necesidad de trabajar con un asistente como Coq. El aprendizaje de este lenguaje de tipos dependientes, basado en la correspondencia de Curry-Howard, fue una tarea muy compleja (que quedó reflejada en la necesidad de contactar con expertos en foros públicos[6], a los que estamos enormemente agradecidos) y que consumió la totalidad del trabajo durante varios meses, derivando en la creación de Koky [49] y en la publicación de un nuevo artículo en el blog de la empresa [85].

Más importante, el esfuerzo invertido en aprender Coq se vio recompensado con la aceptación de [86]. En esta publicación, se formalizaban las intuiciones

[4]Ver *pull requests* #415, #502 y #533 en `https://github.com/julien-truffaut/Monocle/`.

[5]`https://www.reddit.com/r/scala/comments/7swn90/lens_state_is_your_father_and_i_can_prove_it/`

[6]`https://stackoverflow.com/search?q=user:1263978+[coq]`

que teníamos para el caso particular de lens algebras usando Coq de forma muy elegante y se establecían unas bases para seguir formalizando el resto del catálogo de optic algebras. Esta conferencia, sin tener un índice de impacto alto, está muy bien valorada por la comunidad funcional. Es por eso por lo que decidimos enviar nuestra propuesta aquí, ya que nos parecía una buena primera toma de contacto con la academia. El feedback recibido fue bueno, llegando a resultar inspirador para la derivación automática de type classes en Haskell[7] y suscito un gran interés por parte de los creadores del lenguaje de programación *Eta* (Haskell en la JVM) [8].

La validación de las abstracciones experimentales de stateless con ejemplos sencillos y la elegancia de las pruebas formales para el caso particular de lens parecían claros indicadores de que el resto de piezas del puzzle (es decir, la aplicación del patrón al resto de ópticas) también encajarían correctamente. Sin embargo, nos encontramos con un callejón sin salida que nos tendría bloqueados durante varios meses. El principal problema es que no existían representaciones basadas en transformaciones naturales para otras ópticas, por lo que el primer paso sería dar con ellas. Las abstracciones propuestas en la sección 4.2 parecían bastante razonables, pero dar con el conjunto de leyes que las hicieran isomorfas a otras representaciones ya existentes resultó en una tarea especialmente tediosa. De hecho, las leyes que surgían eran numerosas y demasiado específicas; las pruebas formales en Coq para los isomorfismos eran extremadamente largas y complejas. Cuando se cambiaba a un plano genérico la situación era incluso peor, ya que no era posible trasladar muchas de las leyes, muy atadas a las estructuras que la definían. Afortunadamente, la búsqueda de una solución para los problemas de eficiencia mencionados anteriormente nos hizo poner el foco en LINQ, donde empezamos a establecer muchas relaciones con nuestro trabajo. El campo de LINQ abría muchas puertas, lo cual nos permitió salir del fango y empezar una nueva etapa de investigación.

Reorientación hacia *Optica*

Las técnicas que se persiguen con esta noción de LINQ tienen por objetivo acercar los lenguajes de queries a los lenguajes de propósito general, así como derivar interpretaciones óptimas para bases de datos. Esto encajaba muy bien con nuestra idea inicial, lo que nos permitía entender las ópticas como un posible lenguaje para hacer LINQ, que se postulase como una alternativa a las for-comprehensions. El énfasis de este campo por la normalización de las expresiones arrojaba mucha luz sobre los problemas de eficiencia que veníamos experimentando en stateless. El artículo de Cheney *et al.* [19] fue de especial relevancia para obtener esa nueva perspectiva, pero los fundamentos de los *quoted* DSLs en los que se apoyaba nos resultaban un tanto lejanos —a pesar de los conocimientos sobre *macros* adquiridos durante la implementación de updatable [46].

Afortunadamente, descubrimos que Kiselyov *et al.* [114, 73, 68] habían implementado la misma idea siguiendo la aproximación de *tagless-final*, con la que nos encontrábamos muy cómodos, ya que siempre se ha considerado un material esencial en los cursos de programación funcional que se imparten desde Habla

[7] https://twitter.com/Iceland_jack/status/1069071938331582464
[8] https://github.com/rahulmutt/algebraic-optics

Computing. No obstante, estos trabajos evidenciaban un matiz muy sutil pero diferenciador que chocaba con nuestra perspectiva de tagless-final, que hasta entonces estaba muy ligada a MTL: las representaciones no eran computaciones. Bajo esta premisa, comenzamos a desarrollar Optica, con unas primitivas que devolvían ópticas, que se combinaban con otras primitivas que también hacían lo propio. Para nuestra sorpresa, la semántica estándar del lenguaje era simple y elegante, por lo que se comenzó a desarrollar semánticas no estándar para XQuery y SQL. La última de ellas no fue trivial, debido principalmente en nuestro empeño por *normalizar* en lugar de *evaluar*. La impedancia existente entre el modelo anidado en el que se apoyan las ópticas y el modelo relacional de SQL hacían que dicha normalización no tuviera sentido. Entender este aspecto fue fundamental para concluir esta interpretación.

Con los nuevos resultados, se comienza a escribir un *technical report* con el que pretendíamos obtener feedback por parte de otros expertos en el campo de LINQ. Dicho artículo, que postulaba el nuevo lenguaje como una alternativa a las for-comprehensions para LINQ (manteniendo quizás un tono demasiado agresivo hacia éstas), fue enviado a varios expertos, entre ellos Cheney y Kiselyov, a los que agradecemos enormemente el feedback recibido. En particular, Cheney nos invitó a reorientar nuestra investigación hacia una visión más conciliadora con el trabajo existente, mostrándonos cómo integrar nuestra aproximación con su trabajo en [19], lo que derivó en la sección 6.5 de esta tesis. Finalmente, se escribe un artículo académico que se envía a la revista *Science of Computer Programming*, donde los revisores anónimos nos orientan hacia un formato de presentación más formal, que puede apreciarse a lo largo del capítulo 6, que consideramos mejora la calidad de los resultados encarecidamente [87]. El trabajo de investigación cumple con los objetivos de los que partíamos: llevar los beneficios de las ópticas a un plano general que permita la generación de consultas eficientes. Además, abre muchas líneas de investigación, tal y como se muestra a lo largo del capítulo 9.

Bibliografía

[1] Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., and Stevens, P. Notions of bidirectional computation and entangled state monads. In *International Conference on Mathematics of Program Construction* (2015), Springer, pp. 187–214.

[2] Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., and Stevens, P. Reflections on monadic lenses. In *A List of Successes that can Change the World*. Springer, 2016, pp. 1–31.

[3] arrow-kt. Arrow: Functional companion to kotlin's standard library. `https://github.com/arrow-kt/arrow`, 2014. Last checked: 16/01/2020.

[4] Awodey, S. *Category theory*. Oxford University Press, 2010.

[5] Bachman, C. W. The programmer as navigator. *Commun. ACM 16*, 11 (Nov. 1973), 653–658.

[6] Barry, D., and Stanienda, T. Solving the java object storage problem. *Computer 31*, 11 (1998), 33–40.

[7] Benedikt, M., and Cheney, J. Semantics, types and effects for XML updates. In *Database Programming Languages* (Berlin, Heidelberg, 2009), P. Gardner and F. Geerts, Eds., Springer Berlin Heidelberg, pp. 1–17.

[8] Brady, E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming 23*, 5 (2013), 552–593.

[9] Brasil, F. W. Quill - compile-time language integrated queries for Scala. `https://getquill.io`. Last checked: 14/6/2019.

[10] Buneman, P., Libkin, L., Suciu, D., Tannen, V., and Wong, L. Comprehension syntax. *SIGMOD Record 23*, 1 (1994), 87–96.

[11] Buneman, P., Naqvi, S. A., Tannen, V., and Wong, L. Principles of programming with complex objects and collection types. *Theor. Comput. Sci. 149*, 1 (1995), 3–48.

[12] Burmako, E. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala* (New York, NY, USA, 2013), SCALA '13, ACM, pp. 3:1–3:10.

[13] Carette, J., Kiselyov, O., and Shan, C.-c. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming 19*, 5 (2009), 509–543.

[14] Ceri, S., Gottlob, G., and Tanca, L. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering 1*, 1 (1989), 146–166.

[15] Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. Associated types with class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2005), POPL '05, ACM, pp. 1–13.

[16] Cheney, J. Language-integrated query: state of the art and open problems. In *NII Shonan Meeting Report No. 2017-6* (2017).

[17] Cheney, J. Email corrrespondence. Personal communication, May 2019.

[18] Cheney, J., and Hinze, R. First-class phantom types. Tech. rep., Cornell University, 2003.

[19] Cheney, J., Lindley, S., and Wadler, P. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2013), ICFP '13, ACM, pp. 403–416.

[20] Cheney, J., Lindley, S., and Wadler, P. Query shredding: efficient relational evaluation of queries over nested multisets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 1027–1038.

[21] Chiusano, P., and Bjarnason, R. *Functional programming in Scala*. Manning Publications Co., 2014.

[22] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices 46*, 4 (2011), 53–64.

[23] Cooper, E. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *Database Programming Languages - DBPL 2009, 12th International Symposium, Lyon, France, August 24, 2009. Proceedings* (2009), pp. 36–51.

[24] Copeland, G., and Maier, D. Making Smalltalk a database system. In *ACM Sigmod Record* (1984), vol. 14, ACM, pp. 316–325.

[25] Date, C. J. A critique of the sql database language. *SIGMOD Rec. 14*, 3 (Nov. 1984), 8–54.

[26] Dviánszky, P. Lgtk api correction. `https://people.inf.elte.hu/divip/LGtk/CorrectedAPI.html`, April 2013. Last checked: 01/08/2019.

[27] Elliott, C. Semantic editor combinators. `http://conal.net/blog/posts/semantic-editor-combinators`, November 2008. Last checked: 25/11/2019.

[28] Evans, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[29] Fegaras, L. An algebra for distributed big data analytics. *Journal of Functional Programming 27* (2017), e27.

[30] Fernández, M., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W.-C. Silkroute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS) 27*, 4 (2002), 438–493.

[31] Fischer, S., Hu, Z., and Pacheco, H. A clear picture of lens laws. In *International Conference on Mathematics of Program Construction* (2015), Springer, pp. 215–223.

[32] Fluet, M., and Pucella, R. Phantom types and subtyping. *Journal of Functional Programming 16*, 6 (2006), 751–791.

[33] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 29*, 3 (2007), 17.

[34] Fowler, M. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[35] Ghosh, D. *DSLs in action*. Manning Publications Co., 2010.

[36] Ghosh, D. *Functional and reactive domain modeling*. Manning Publications Company, 2016.

[37] Gibbons, J. Unifying theories of programming with monads. In *International Symposium on Unifying Theories of Programming* (2012), Springer, pp. 23–67.

[38] Gibbons, J., Henglein, F., Hinze, R., and Wu, N. Relational algebra by way of adjunctions. *PACMPL 2*, ICFP (2018), 86:1–86:28.

[39] Gibbons, J., and Hinze, R. Just do it: simple monadic equational reasoning. *ACM SIGPLAN Notices 46*, 9 (2011), 2–14.

[40] Grabmüller, M. Monad transformers step by step. *Draft paper, October* (2006).

[41] Grenrus, O. Affine traversals. `http://oleg.fi/gists/posts/2017-03-20-affine-traversal.html`, March 2017. Last checked: 13/11/2019.

[42] Grenrus, O. Indexed profunctor optics. `http://oleg.fi/gists/posts/2017-04-26-indexed-poptics.html`, April 2017. Last checked: 16/08/2019.

[43] Grenrus, O. Glassery. `http://oleg.fi/gists/posts/2017-04-18-glassery.html`, Apr 2018. Last checked: 14/6/2019.

[44] Grust, T., Rittinger, J., and Schreiber, T. Avalanche-safe linq compilation. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 162–172.

[45] Gurnell, D. *The Type Astronaut's Guide to Shapeless*. Lulu. com, 2017.

[46] Habla Computing SL. Updatable: updating immutable objects in generic contexts. `https://github.com/hablapps/updatable`, 2013. Last checked: 16/01/2020.

[47] Habla Computing SL. Coalgebraz: fun with coalgebras! `https://github.com/hablapps/coalgebraz`, 2016. Last checked: 16/01/2020.

[48] Habla Computing SL. Stateless: a library which is not recognized as citizen of any infrastructure. `https://github.com/hablapps/stateless`, 2017. Last checked: 12/08/2019.

[49] Habla Computing SL. Koky: type classes, datatypes and theorems for functional programming in coq. `https://github.com/hablapps/koky`, 2018. Last checked: 20/01/2020.

[50] Habla Computing SL. Lens algebra. `https://github.com/hablapps/lensalgebra`, 2018. Last checked: 20/01/2020.

[51] Habla Computing SL. Naturally: programming in a tagless-final style, naturally. `https://github.com/hablapps/naturally`, 2018. Last checked: 17/12/2019.

[52] Habla Computing SL. Optica: Optic-based language-integrated query. `https://github.com/hablapps/scico19`, 2019. Last checked: 13/11/2019.

[53] Han, J., Haihong, E., Le, G., and Du, J. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications* (2011), IEEE, pp. 363–366.

[54] Hartig, O., and Pérez, J. An initial analysis of Facebook's GraphQL language. In *AMW 2017 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017.* (2017), vol. 1912, Juan Reutter, Divesh Srivastava.

[55] Haskell Wiki. Monad transformer library. `https://wiki.haskell.org/Monad_Transformer_Library`, December 2012. Last checked: 05/09/2019.

[56] Hinze, R., et al. Fun with phantom types. *The fun of programming* (2003), 245–262.

[57] Hofmann, M., Pierce, B., and Wagner, D. Symmetric lenses. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 371–384.

[58] Horn, R., Perera, R., and Cheney, J. Incremental relational lenses. *Proc. ACM Program. Lang. 2*, ICFP (July 2018), 74:1–74:30.

[59] Hudak, P. Functional reactive programming. In *Programming Languages and Systems* (Berlin, Heidelberg, 1999), S. D. Swierstra, Ed., Springer Berlin Heidelberg, pp. 1–1.

[60] Hudak, P., et al. Building domain-specific embedded languages. *ACM Comput. Surv. 28*, 4es (1996), 196.

[61] Hughes, J. Why functional programming matters. *The computer journal 32*, 2 (1989), 98–107.

[62] Hutton, G. Fold and unfold for program semantics. *SIGPLAN Not. 34*, 1 (Sept. 1998), 280–288.

[63] Ireland, C., Bowers, D., Newton, M., and Waugh, K. A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications* (2009), IEEE, pp. 36–43.

[64] Jacobs, B. Objects and classes, coalgebraically. In *Object-Orientation with Parallelism and Persistence* (1995), Kluwer Acad. Publ, pp. 83–103.

[65] Jacobs, B., and Rutten, J. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science 62* (1997), 222–259.

[66] Jones, M. P. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming* (1995), Springer, pp. 97–136.

[67] Jones, M. P. Type classes with functional dependencies. In *European Symposium on Programming* (2000), Springer, pp. 230–244.

[68] Katsushima, T., and Kiselyov, O. Language-integrated query with ordering, grouping and outer joins (poster paper). In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017* (2017), pp. 123–124.

[69] Keller, W. Mapping objects to tables. In *Proc. of European Conference on Pattern Languages of Programming and Computing, Kloster Irsee, Germany* (1997), vol. 206, Citeseer, p. 207.

[70] Kiselyov, O. Typed tagless final interpreters. In *Generic and Indexed Programming*. Springer, 2012, pp. 130–174.

[71] Kiselyov, O. Effects without monads: Non-determinism back to the meta language. Tech. rep., 2017.

[72] Kiselyov, O., et al. Reconciling abstraction with high performance: A metaocaml approach. *Foundations and Trends® in Programming Languages 5*, 1 (2018), 1–101.

[73] Kiselyov, O., and Katsushima, T. Sound and efficient language-integrated query - maintaining the ORDER. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings* (2017), pp. 364–383.

[74] Kleppmann, M. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[75] Kmett, E. lens: Lenses, folds and traversals. `https://github.com/ekmett/lens`, 2012. Last checked: 25/11/2019.

[76] Koch, C. Abstraction without regret in database systems building: a manifesto. *IEEE Data Engineering Bulletin 37*, ARTICLE (2014).

[77] Krötzsch, M., Simancik, F., and Horrocks, I. A description logic primer. *CoRR abs/1201.4089* (2013).

[78] Lämmel, R., and Meijer, E. Revealing the x/o impedance mismatch. In *International Spring School on Datatype-Generic Programming* (2006), Springer, pp. 285–367.

[79] Le, J. mtl is not a monad transformer library. `https://blog.jle.im/entry/mtl-is-not-a-monad-transformer-library.html`, May 2015. Last checked: 05/09/2019.

[80] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), ACM, pp. 333–343.

[81] Lipovaca, M. *Learn you a haskell for great good!: a beginner's guide.* no starch press, 2011.

[82] López-González, J., and Serrano, J. M. Lens, state is your father. `https://blog.hablapps.com/2016/11/10/lens-state-is-your-father/`, November 2016. Last checked: 20/01/2020.

[83] López-González, J., and Serrano, J. M. Yo dawg, we put an algebra in your coalgebra. `https://blog.hablapps.com/2016/10/11/yo-dawg-we-put-an-algebra-in-your-coalgebra/`, October 2016. Last checked: 19/01/2020.

[84] López-González, J., and Serrano, J. M. Don't fear the profunctor optics. `https://github.com/hablapps/DontFearTheProfunctorOptics`, September 2017. Last checked: 07/08/2019.

[85] López-González, J., and Serrano, J. M. Lens, state is your father... and i can prove it! `https://blog.hablapps.com/2018/01/24/lens-state-is-your-father-and-i-can-prove-it/`, January 2018. Last checked: 20/01/2020.

[86] López-González, J., and Serrano, J. M. Towards optic-based algebraic theories: The case of lenses. In *Trends in Functional Programming* (Cham, 2019), M. Pałka and M. Myreen, Eds., Springer International Publishing, pp. 74–93.

[87] López-González, J., and Serrano, J. M. The optics of language-integrated query. *Science of Computer Programming* (2020), 102395.

[88] Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D. Understanding the use of lambda expressions in java. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 85.

[89] McBride, C., and Paterson, R. Applicative programming with effects. *Journal of functional programming 18*, 1 (2008), 1–13.

[90] Meijer, E., Beckman, B., and Bierman, G. M. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006* (2006), p. 706.

[91] Milewski, B. From lenses to yoneda embedding. `https://bartoszmilewski.com/2015/07/13/from-lenses-to-yoneda-embedding/`, July 2015. Last checked: 25/11/2019.

[92] Moggi, E. Notions of computation and monads. *Information and computation 93*, 1 (1991), 55–92.

[93] Najd, S., Lindley, S., Svenningsson, J., and Wadler, P. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (2016), ACM, pp. 25–36.

[94] Norris, R. Doobie - a functional JDBC layer for Scala. `https://tpolecat.github.io/doobie/`. Last checked: 14/6/2019.

[95] O'Connor, R. Functor is to lens as applicative is to biplate: Introducing multiplate. In *7th ACM SIGPLAN Workshop on Generic Programming* (2011), ACM.

[96] O'Connor, R. mezzolens:pure profunctor functional lenses. `http://hackage.haskell.org/package/mezzolens`, 2011. Last checked: 27/11/2019.

[97] O'Connor, R. Polymorphic update with van laarhoven lenses. `http://r6.ca/blog/20120623T104901Z.html`, June 2012. Last checked: 21/11/2019.

[98] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. An overview of the Scala programming language. Tech. rep., 2004.

[99] Odersky, M., Petrashko, D., Martres, G., et al. The Dotty project.

[100] Odersky, M., Spoon, L., and Venners, B. *Programming in scala*. Artima Inc, 2008.

[101] Oliveira, B. C., Moors, A., and Odersky, M. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 341–360.

[102] Oliveira, B. C. d. S., and Cook, W. R. Extensibility for the masses. In *European Conference on Object-Oriented Programming* (2012), Springer, pp. 2–27.

[103] Pacheco, H., Hu, Z., and Fischer, S. Monadic combinators for putback style bidirectional programming. In *Proceedings of the acm sigplan 2014 workshop on partial evaluation and program manipulation* (2014), ACM, pp. 39–50.

[104] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972), 1053–1058.

[105] Pickering, M., Wu, N., and Gibbons, J. Profunctor optics: Modular data accessors. *Art, Science, and Engineering of Programming 1*, 2 (4 2017).

[106] Pierce, B. C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* (2010).

[107] purescript-contrib. purescript-profunctor-lenses. `https://github.com/purescript-contrib/purescript-profunctor-lenses`, 2015. Last checked: 27/11/2019.

[108] Rompf, T., and Odersky, M. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (New York, NY, USA, 2010), GPCE '10, Association for Computing Machinery, p. 127–136.

[109] Serrano, J. M., and Saugar, S. Operational semantics of multiagent interactions. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems* (2007), pp. 1–8.

[110] Serrano, J. M., Saugar, S., Laurendi, R., and Buccafurri, F. A contextual reading of conditional commitments. In *ECAI* (2010), pp. 993–994.

[111] Shkaravska, O. Side-effect monad, its equational theory and applications. Arvutiteaduse teooriaseminar, 2005.

[112] Sozeau, M., and Oury, N. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics* (2008), Springer, pp. 278–293.

[113] Spolsky, J. The law of leaky abstractions. In *Joel on Software*. Springer, 2004, pp. 197–202.

[114] Suzuki, K., Kiselyov, O., and Kameyama, Y. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (New York, NY, USA, 2016), PEPM '16, ACM, pp. 37–48.

[115] Syme, D. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006* (2006), pp. 43–54.

[116] Taha, W. Domain-specific languages. In *ICCES'08. International Conference on Computer Engineering & Systems, Cairo, Egypt, 25-27 November* (2008), IEEE Press, pp. XXV–XXVIII.

[117] Tannen, V., Buneman, P., and Wong, L. Naturally embedded query languages. In *Database Theory - ICDT'92, 4th International Conference, Berlin, Germany, October 14-16, 1992, Proceedings* (1992), pp. 140–154.

[118] Trinder, P., and Wadler, P. Improving list comprehension database queries. In *Fourth IEEE Region 10 International Conference TENCON* (Nov 1989), pp. 186–192.

[119] Truffaut, J. Monocle: optics library for Scala. `https://github.com/julien-truffaut/Monocle`, 2014. Last checked: 13/08/2019.

[120] Truffaut, J. Email corrrespondence. Personal communication, 2020.

[121] Turner, D. A. Total functional programming. *J. UCS 10*, 7 (2004), 751–768.

[122] Ulrich, A., Giorgidze, G., Weijers, J., and Schweinsberg, N. DSH: Database supported Haskell. `http://hackage.haskell.org/package/DSH`, 2000–2004.

[123] van Laarhoven, T. Cps based functional references. `https://twanvl.nl/blog/haskell/cps-functional-references`, July 2009. Last checked: 20/11/2019.

[124] van Laarhoven, T. Isomorphism lenses. `https://twanvl.nl/blog/haskell/isomorphism-lenses`, May 2011. Last checked: 06/08/2019.

[125] van Laarhoven, T. Lenses: viewing and updating data structures in Haskell. Foundations Seminar, May 2011.

[126] Wadler, P. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (1990), ACM, pp. 61–78.

[127] Wadler, P. Monads for functional programming. In *International School on Advanced Functional Programming* (1995), Springer, pp. 24–52.

[128] Wadler, P. XQuery: A typed functional language for querying XML. In *International School on Advanced Functional Programming* (2002), Springer, pp. 188–212.

[129] Wadler, P., and Blott, S. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), ACM, pp. 60–76.

[130] Wong, L. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci. 52*, 3 (1996), 495–505.

[131] Wong, L. Kleisli, a functional query system. *J. Funct. Program. 10*, 1 (Jan. 2000), 19–56.

[132] Wright, G. *Academia obscura: The hidden silly side of higher education.* Unbound Publishing, 2017.

[133] Xi, H., Chen, C., and Chen, G. Guarded recursive datatype constructors. *SIGPLAN Not. 38*, 1 (Jan. 2003), 224–235.