Optic-Based Language-Integrated Query

J. LÓPEZ-GONZÁLEZ, Universidad Rey Juan Carlos and Habla Computing, S.L
JUAN M. SERRANO, Universidad Rey Juan Carlos and Habla Computing, S.L

Comprehensions reign over the realm of language-integrated query (LINQ). Indeed, to our knowledge, all the theories and systems for LINQ so far have surrendered to the monadic perspective in order to tackle the integration of external query languages into a host language. This paper puts forward a new contender in the field, which champions as the technique of choice to build modular data accessors for immutable structures: optics. In order to enable the use of optics for LINQ, they must be first lifted into a full-blown DSL. We show how to do that for a restricted subset of optics, namely getters, affine folds and folds. The type system of the resulting *language of optics* distills their compositional properties, whereas its denotational semantics is given by standard optics themselves. We formalize and implement the language as a typed tagless-final representation in Scala. Then, we reimplement several common queries from the LINQ literature to show the gains in readability and reusability with respect to their comprehension-based counterparts. Last, we successfully challenge the effectiveness of the language of optics for LINQ with the design of two non-standard optic representations: one which allows us to derive XQuery expressions, and another one for SQL queries. In sum, this work lays the foundation stone for an industry-oriented optic library which adopts the dozens of combinators and the many kinds of optics that one can find in the folklore.

Additional Key Words and Phrases: optics, language-integrated query, domain-specific language

ACM Reference Format:
J. López-González and Juan M. Serrano. 2019. Optic-Based Language-Integrated Query. 1, 1 (April 2019), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The purpose of language-integrated query (LINQ) is achieving a smooth integration between query languages, on the one hand, and general-purpose host languages, on the other. In order to do so, the query language needs to be embedded within the host language, which may be achieved through different techniques such as typed tagless-final [4, 16], GADTs [6] or Quoted DSLs [21].

To our knowledge, however, regardless of the particular technique to create such an embedded DSL (EDSL) [15], all the approaches to LINQ so far has tackled this integration via comprehensions [27], i.e. from a monadic perspective [28]. The basic insight was originally introduced in [26], and then exploited by different theories and systems: NRC [2, 3], LINKS [8], Microsoft's LINQ [20], T-LINQ [7], QUEA [25], SQUR [18].

In essence, the idea is borrowing the comprehension syntax that lists, sets, bags, and other bulk types enjoy, to express queries at a generic monadic level. These queries can then be interpreted over external querying systems, such as relational databases or NoSQL stores. For instance, consider the data structures in column "comprehensions" of Table 1. As you can see, couples point at people by means of keys, according to a flat model. Now, given a list of couples and a list of people, we may obtain the names of those female partners who are under 50 years of age using a list comprehension, as query `under50_a` shows.

XXXX-XXXX/2019/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

	comprehension	*optics*
model	`class Couple(her: String, him: String)` `class Person(name: String, age: Int)`	`class Couple(her: Person, him: Person)` `class Person(name: String, age: Int)`

immutable	`def under50_a(`
	`  couples: List[Couple],`
	`  people: List[Person]): List[String] =`
	`  for {`
	`  c ← couples`
	`  w ← people`
	`  if c.her == w.name && w.age < 50`
	`  } yield w.name`
		`val under50_c: Fold[List[Couple], String] =`
		`  couples ▹ her ▹ filtered (age < 50) ▹ name`

generic	`val under50_b =`
	`  quote {`
	`  for {`
	`  c ← query[Couple]`
	`  w ← query[Person]`
	`  if c.her == w.name && w.age < 50`
	`  } yield w.name`
	`  }`
		?

Table 1. Towards Optic-Based LINQ.

Now, using Quill[1], a quoted DSL in Scala heavily inspired by T-LINQ [7, 21], we may abstract away any reference to lists and express the very same query in a generic way. You can find it in the "comprehension" column of the aforementioned table, named `under50_b`. This query may be compiled to different targets according to the mapping between Scala case classes and the database schemas that the Quill framework supports (as of this writing Cassandra's CQL [11] and SQL). For instance, the resulting SQL expression generated for this query would be:

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

This paper aims to challenge the status quo of comprehensions and proposes LINQ to build upon a different foundation, namely optics [24]. The idea is similar: using lenses [10], traversals [22], folds, and other optic abstractions, we can query and update immutable data structures quite naturally. Why then not borrowing this very same syntax to express generic queries that can be interpreted over different target storage systems (immutable structures being one of them)? We foresee several advantages of using optics for LINQ:

- Optics favour a hierarchical modeling of data, which leads oftentimes to simpler queries. This is evidenced in column "optics" of Table 1, where keys in models are replaced by full-blown values. Arguably, query `under50_c`, which is compiled using several optics (the fold `couples`, and getters `her`, `age` and `name`), is more simple and readable that the corresponding query `under50_b`. This is mainly due to the fact that the comprehension approach for LINQ is eminently relational, forcing the programmer to take care of keys in query definitions. Moreover, this does not only have a penalty in the readability of queries, but, more importantly, in the possibility of reusing standard combinators. To overcome this problem, some authors propose to translate flat representations into nested ones manually, write queries over the nested representation, and reverse the process automatically through a complex rewriting machinery [7]. None of this complexity would be required while using optics for writing query definitions.
- Optics are known as *functional references*, i.e. structures that point at parts which are contextualized within a whole, and provide methods to access *and* update the values they are

pointing at. Moreover, optics excel at compositionality, with plenty of combinators to build queries out of basic primitives pointing at single, optional or multiple foci. In contrast, the standard semantics of LINQ-based comprehensions corresponds to a multiset [18] that denotes the collection of elements referred to by the query. So, queries are explicitly tied to a *retrieving* operation, leaving aside from the theory how to formalize other equally necessary operations such as *updates*. The declarative approach to query composition favoured by optics is presumably more apt to cope with this problem.

- Even if we stick to retrieving queries, we find a lack of specialised abstractions while composing queries, beyond multisets (or lists as they are usually represented). For instance, take *getAge* from [7] as an example, which takes the name of a person as parameter and outputs her age, as long as the person is found in the store. Considering that names are keys, one would expect at most one integer as result. However, the original query declares a list of integers as result type, since it is tied to list-comprehensions. Optics provide a richer type system to model queries in a more precise way.

This paper paves the way to the use of optics as a first-class language for LINQ. In particular, these are its contributions:

- We review standard optics from the point of view of LINQ, and show how to exploit a subset of standard optic abstractions (namely, getters, affine folds and folds) and their combinators to express compositional queries (Sect. 2).
- We lift standard optics into a full-fledged DSL. In particular, we introduce *Stateless*, a language whose syntax and type system is defined in close correspondence with optics, and whose denotational semantics are given by standard optics themselves (Sect. 3). We show how to implement generic queries over generic models using Stateless in a declarative way, once and for all, so that they can be later reused to target different querying infrastructures and representation formats, including but not restricted to immutable data structures.
- We challenge this feature of Stateless as a general query language with the implementation of two interpreters: one that allows us to translate Stateless queries to XQuery expressions (Sect. 4), and another one that targets SQL statements (Sect. 5). As we will see, the XQuery interpreter can be given in a purely compositional way, since XML documents are nested structures. In the case of the SQL interpreter, we need to adapt the optic model into the relational one with the help of an intermediate language.

As you can see, Sect. 2, Sect. 3, Sect. 4 and Sect. 5 contain the bulk of the paper. Section 6 discusses related work and limitations of the approach. Section 7 concludes and points towards current and future work.

2 QUERYING WITH OPTICS

In the context of this paper, we will focus exclusively on those optics which are able to access the parts from the whole but do not write back in the data structure. These optics are not as widespread as their siblings with updating capabilities, since accessing nested fields from immutable data structures is usually a trivial task. Nonetheless, they exhibit the same compositional features and patterns as the rest of optics, as shown along this section. We will first introduce the definitions of *getters*, *affine folds* and *folds*[1], together with their combinators. Then, we will illustrate the declarative querying style advocated by optics with two examples that will be used throughout the paper.

[1]We also ignore other read-only optics such as *fold1*, since they do not add value in the particular examples that we have selected for this paper.

2.1 Getters, Affine Folds and Folds

First of all, we adopt a *concrete* optic representation, where the notion of *whole* and *part* is evident, to make definitions more understandable. There are other representations, such as *van Laarhoven* [22] or *profunctor* [24], where optic composition is implemented in a remarkably elegant way, whose signatures, however, are not as easy to approach for an outsider[2].

Definition 2.1 (Getter). A getter wraps a function from the whole to a single part. We could see this optic as a simplification of a *lens*, where we get rid of the updating part. We encode it in Scala as follows:

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

The type parameters S and A will consistently serve as the whole and the part, respectively, throughout the different optic definitions. Moreover, the field which stores the viewing function will be named *run*, to emphasise the declarative nature of optics composition.

As mere function wrappers, getters conform a category with products[3]

```
implicit object GetterCat extends CategoryWithProduct[Getter] {
  def id[A]: Getter[A, A] = Getter(identity)
  def andThen[A, B, C](f: Getter[A, B], g: Getter[B, C]): Getter[A, C] =
    Getter(f.run ▹ g.run)
  def fork[S, A, B](f: Getter[S, A], g: Getter[S, B]): Getter[S, (A, B)] =
    Getter(s => (f.run(s), g.run(s)))
}
```

We can see `id` as a getter where source and target do coincide, neutral under composition. The `andThen` method allows us to combine getters which are pointing at nested values. Finally, `fork` is required if we want to put different targets together. We informally refer to this way of composing as *horizontal* composition.

We assume ▹ and ▵ as infix versions of the categorical methods `andThen` and `fork`, respectively, where the last symbol has precedence over the first one. In fact, we have already used this notation in the implementation of `andThen`, where ▹ refers to the category of types and plain functions.

Horizontal optic composition is not widespread in the folklore. Indeed, it is not possible to implement it in a safe way for most optics. For example, an analogous implementation for composing lenses horizontally would violate lens laws [9] when both getters point at the very same focus. In this sense, we must say that the optics that we show in this section are lawless, since they do not need to keep consistency between views and updates, and therefore they are freer to enable new ways of composition.

There are several getter expressions that will be used recurrently along the article, that we present here.

```
object Getter {
  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 eq[S, A: Equal](x: Getter[S, A], y: Getter[S, A]): Getter[S, Boolean] =
    x ▵ y ▹ Getter(_ === _)
  def gt[S](x: Getter[S, Int], y: Getter[S, Int]): Getter[S, Boolean] =
    x ▵ y ▹ Getter(_ > _)
  def sub[S](x: Getter[S, Int], y: Getter[S, Int]): Getter[S, Int] =
    x ▵ y ▹ Getter(_ - _)
}
```

[2] As evidenced by the jokes around this topic in the functional programming community https://pbs.twimg.com/media/CypY7B1W8AAvqwl.jpg

[3] We have ignored projections in the product definition, since we will not use them in this paper.

The function `like` grants access to a constant target, where the source is ignored. Then we find functions related to arithmetic operations that take getters pointing at operands as parameters. Essentially, these operands are composed with the lifting of plain operators into the category of getters[4]. We will assume the operators `===`, `>` and `-` as infix versions of `eq`, `gt` and `sub`, respectively.

Definition 2.2 (AffineFold). An affine fold wraps a function from the whole to an optional part. We encode it as follows:

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

We could see this optic as a simplification of an affine traversal, where we get rid of the updating part.

As any other optic, affine folds conform a category:

```
implicit object AffineFoldCat extends Category[AffineFold] {
  def id[A]: AffineFold[A, A] = AffineFold(ask[Option, A])
  def andThen[A, B, C](f: AffineFold[A, B],
                       g: AffineFold[B, C]): AffineFold[A, C] =
    AffineFold(kleisli(f.run) ▹ kleisli(g.run))
}
```

It is worth mentioning that `ask` provides the identity kleisli function for `Option`, i.e. it simply wraps the argument in a `Some` case. If we pay attention to `andThen`, we can see that its implementation is basically kleisli composition. Thereby, if any target in the composition chain is empty, the resulting affine fold will point to an empty target as well.

Now, consider *filtered* as an interesting builder of affine folds:

```
object AffineFold {
  def filtered[S](p: Getter[S, Boolean]): AffineFold[S, S] =
    AffineFold(s => if (p.run(s)) Some(s) else None)
}
```

Notice that this optic declares the same types for whole and part. It just discards all those values which do not hold for the input predicate, by returning a `None` value. In addition, it is worth emphasizing that the predicate is a getter itself. This is not widespread in folklore libraries, where a plain lambda predicate is taken as argument instead. However, predicates can be perfectly understood as queries (getters, in particular), and this will lead to a simpler DSL in the next section where lambdas need not be taken into account.

Definition 2.3 (Fold). A fold is an optic that we can use to point at a (possibly empty) sequence of parts contextualized in a whole.

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

We could see this optic as a simplification of a traversal [22], where we get rid of the updating part.

As usual, we present the category instance for folds

```
implicit object FoldCat extends Category[Fold] {
  def id[A]: Fold[A, A] = Fold(ask[List, A])
  def andThen[A, B, C](f: Fold[A, B], g: Fold[B, C]): Fold[A, C] =
    Fold(kleisli(f.run) ▹ kleisli(g.run))
}
```

Notice that this implementation is basically the same as the one we showed for affine folds, but here we are working with kleislis for `List`. In fact, we could have provided the same pattern for getters, using kleislis for `Id` but we have decided to adopt the standing implementation for simplicity.

We introduce typical fold-related combinators that will be used in the guiding examples later on.

[4]Note that we may also have introduced definitions for the lifted operators themselves.

```
object Fold {
  def nonEmpty[S, A](fl: Fold[S, A]): Getter[S, Boolean] =
    Getter(fl.run(_).nonEmpty) /* List.nonEmpty */
  def empty[S, A](fl: Fold[S, A]): Getter[S, Boolean] =
    not(nonEmpty(fl))
  def all[S, A](fl: Fold[S, A])(p: Getter[A, Boolean]): Getter[S, Boolean] =
    empty(fl ▹ filtered(not(p)))
  def any[S, A](fl: Fold[S, A])(p: Getter[A, Boolean]): Getter[S, Boolean] =
    not(all(fl)(not(p)))
  def elem[S, A: Equal](fl: Fold[S, A])(a: A): Getter[S, Boolean] =
    any(fl)(id === like(a))
}
```

Broadly speaking, these methods provide standard functionality to deal with a sequence of foci, turning folds into getter predicates. We can use them to check if the number of foci is zero or not, if all (or any) target hold a given predicate and we can query if certain value is contained among the elements of the foci. Beyond the richness of combinators, there is an interesting optic feature which is evidenced in `all`: we are combining a fold with an affine fold to obtain a new fold.

Indeed, one of the major benefits of optics is that they compose heterogeneously[5]. Thereby, it is possible to combine getters, affine folds an folds. Put more simply, we can turn a getter into an affine fold and an affine fold into a fold. The translation is trivial and can be found in the accompanying sources. However, we assume that this adaptation is implicit for simplicity. We will also assume object-oriented notation for these methods, which is idiomatic in Scala. For instance, `nonEmpty(fl)` becomes `fl.nonEmpty` and `all(fl)(p)` becomes `fl.all(p)`.

The implementation of `elem` is also interesting. Since we favour getters over plain functions as predicates, we use optic abstractions and combinators to build our queries. By adopting this first-order style (i.e. no lambdas allowed), we do not have an explicit reference to the predicate parameter, which is required to do the comparison. Fortunately, we can circumvent this problem by using `id`, that we can see as a getter which points from the whole at itself. If we use `like` to wrap `a` we can invoke `eq` to carry out the comparison.

Now that we have seen many standard combinators and some interesting features from optics, we will show a pair of examples to exercise them and introduce common patterns.

2.2 Application Examples

We have selected two examples from [7] to be used throughout the paper. This will allow us to compare more easily the for-comprehension approach advocated in that paper with ours, based on optics.

2.2.1 Couple Example. The first example deals with a simple relation of couples, where we supply the name and age of each person conforming them.

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

Note that we adopt a nested model to define the associated data structures. Once we have defined them, we provide specific optics which point at their parts.

```
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)
```

[5]With a very few exceptions, which are outside the scope of this paper.

```
}
```

Basically, and for this particular example, we need a getter for each field, where whole and part correspond to data and field types, respectively. There is also a simple fold that we can use to point at each couple from `Couples`, that we see as the root of the nested model.

Now, we can use the standard optics defined in the previous section and the specific optics defined for this domain to conform queries. For instance, we can select the name and age difference of all those women who are older than their mates.

```
val differences: Fold[Couples, (String, Int)] =
  couples ▹ filtered((her ▹ age) > (him ▹ age))
          ▹ (her ▹ name) Δ ((her ▹ age) - (him ▹ age))
```

Firstly, we use `couples` as an entry point and we use `filtered` to remove the couples where her age in not greater than his age. Right after filtering, we select her name and we put it together with the age difference, by means of `Δ`, to determine the query output.

The type of the query expression is `Fold[Couples, (String, Int)]`. If you recall the definition of fold, we could *execute* this query by invoking its `run` method, passing a list of couples as argument. Again, we have selected the same data that was introduced in the original paper, so the same result is expected.

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

The comment in the last line of the snippet shows the value that we get in `res` when we run the query. As expected, it indicates that Alex and Cora are older than their mates by 5 and 2 years, respectively.

2.2.2 Organization Example. Now, our model is an organization which is conformed by employees. In addition, each employee has a set of tasks that he is able to do.

```
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)
```

Again, we should supply specific optics for this domain.

```
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)
}
```

In this case, we find several fields containing lists, so we provide folds instead of getters, which are better suited to deal with multiple foci. Now, we will define a query to output the name of those departments where all employees are able to carry out certain task.

```
def expertise(u: Task): Fold[Org, String] =
  departments ▹ filtered(employees.all(tasks.elem(u))) ▹ dpt
```

We find this query easy to read. We refer to all departments and we filter the ones where all employees contain the task `u`. Finally, we supply data to run the query.

```
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("abstract").run(data)
// res: List[String] = List(Quality, Research)
```

The resulting value is telling us that the departments of *Quality* and *Research* are the only ones where all employees are able to *abstract*.

The general pattern should be clear now. Firstly, we define the involved data types in the model and supply specific optics to point at their parts. Secondly, we use these optics and the standard ones to express queries in a modular and elegant way, which are essentially new optics conformed by simpler ones. Finally, we run the query with data to extract the result. As you can see, the approach is eminently declarative: the aspects of building the query and running it are completely separated.

3 STATELESS CORE

Stateless is a domain-specific language that aims at taking optics and their design patterns to a higher level of abstraction. Indeed, its main goal is to allow the definition of generic queries which can be compiled over different querying infrastructures, like XQuery and SQL. This section introduces the core aspects of the language.

Firstly, we will present the syntax and type system of Stateless, where standard primitives and combinators are declared. Secondly, we will show how to provide a generic version of the models and queries that we saw in Sect. 2.2. Finally, we will introduce the standard semantics, that we can use as an interpretation to deal with immutable data structures, i.e. they recover concrete optics.

3.1 Syntax and Type System

We introduce the syntax and type system of Stateless through its embedding in a Scala type class [23] named `Symantics`, following the *typed tagless final* style [5, 16]. We have divided this class into different submodules, one for each kind of optic, to keep the same structure that we followed in the previous section, that we have collected in Fig. 1.

In accordance with the finally tagless style, these modules are parameterized with a type constructor `Repr`, which serves as the generic representation of optic types. The combinators and their syntax are defined in close correspondence with their concrete counterparts introduced in the last section. The only difference is that they do not receive and return plain concrete optics types, but their representations. Concrete types thus aid in the definition of the type system of the language. Typically, they will behave as *phantom types* in most interpretations (but in its denotational semantics, as shown in section 3.3).

We start with `Getters`, where we have applied the representation notion from the previous paragraph to the getter-related combinators. Note that `like` constrains its type parameter `A` to be of a `Base` type, unlike its concrete declaration. This type is an enumeration, currently representing either an integer, string or boolean type. This constraint basically allows us to avoid repeating the

```
trait Getters[Repr[_]] {
  def id_gt[S]: Repr[Getter[S, S]]
  def comp_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 eq[S, A: Base](x: Repr[Getter[S, A]],
                     y: Repr[Getter[S, A]]): Repr[Getter[S, Boolean]]
  def gt[S](x: Repr[Getter[S, Int]],
            y: Repr[Getter[S, Int]]): Repr[Getter[S, Boolean]]
  def sub[S](x: Repr[Getter[S, Int]],
             y: Repr[Getter[S, Int]]): Repr[Getter[S, Int]]
}

trait AffineFolds[Repr[_]] {
  def id_af[S]: Repr[AffineFold[S, S]]
  def comp_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 Folds[Repr[_]] { this: Getters[Repr] with AffineFolds[Repr] =>
  def id_fl[S]: Repr[Fold[S, S]]
  def comp_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 Symantics[Repr[_]] extends Getters[Repr]
    with AffineFolds[Repr] with Folds[Repr] {
  def empty[S, A](fl: Repr[Fold[S, A]]): Repr[Getter[S, Boolean]] =
    not(nonEmpty(fl))
  def any[S, A](fl: Repr[Fold[S, A]])(
                p: Repr[Getter[A, Boolean]]): Repr[Getter[S, Boolean]] =
    nonEmpty(comp_fl(fl, as_fl(filtered(p))))
  def all[S, A](fl: Repr[Fold[S, A]])(
                p: Repr[Getter[A, Boolean]]): Repr[Getter[S, Boolean]] =
    empty(comp_fl(fl, as_fl(filtered(not(p)))))
  def elem[S, A: Base](fl: Repr[Fold[S, A]])(a: A): Repr[Getter[S, Boolean]] =
    any(fl)(eq(id_gt, like(a)))
}
```

Fig. 1. Stateless Symantics.

same lifting operation for the different base types of Stateless[6]. As we will see, we use `like` as the mechanism to represent literals in the language. By doing so, we avoid introducing basic types as semantic domains.

Secondly, we move on to affine fold related primitives. As expected, it contains the categorical primitives for this kind and `filtered`. Its adaptation should be straightforward now. In addition, we supply $\texttt{as}_{af}$ which turns getter representations into affine fold ones (this conversion function was used implicitly in the last section).

[6]This restriction may be lifted in Scala using an alternative representation based on GADTs, combined with partial interpreters implemented as type-level functions that receive the `Base` witness through implicits.

Lastly, we show the fold related combinators. Beyond the categorical primitives, we find `nonEmpty` and $\texttt{as}_{fl}$. You might be missing several combinators that were introduced in Sect. 2 that should belong to this section. However, since they refer to several optic types we implement them as derived combinators in the final `Symantics` trait. As you can see, this trait is conformed by extending the rest of modules.

In the sequel, we make the same syntactic assumptions that were established for plain optics. Namely, we provide infix syntax for binary operations and we assume implicit castings among optics. In addition, we supply *dot* syntax to make our expressions as close as possible to the ones we produce with plain optics. For instance, `all(fl)(p)` becomes `fl.all(p)`.

As a final remark, note that this work focuses on generating representations of queries for different infrastructures, i.e. their execution is beyond the scope of the paper. Thereby, and unlike [25], we do not supply language primitives or types to *observe* the result that we should get by running the generated query, as is customary in the tagless final style.

3.2 Generic Models and Queries

In Sect. 2, we defined case classes and specific optics to model the couple and organization examples. Now, we want to do the same, but in a general way. To do so, we provide a type class which is parameterized over the representation. This is the result that we get for the couple example.

```
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]]
}
```

As you can see, this class generalises the concrete version of `CoupleModel` described in the last section. We will recover that concrete version as a particular interpretation, as will be shown in section 3.3. Also, note that this class makes reference to the domain case classes of `Couple`, `Person`, etc. The role these case classes play in this context is similar to the one played by concrete optic types: they are part of the type system, and will behave as phantom types in most interpretations.

Once we have `Symantics`, where we place the standard optics and combinators, and the data model, where we can find the structure of domain entities in terms of specific optics, we should be able to provide domain queries.

```
def differences[Repr[_] : Symantics : CoupleModel]: Repr[Fold[Couples, (String, Int)]] =
  couples ▹ filtered((her ▹ age) > (him ▹ age))
          ▹ (her ▹ name) △ ((her ▹ age) - (him ▹ age)))
```

The implementation of the generic version of `differences` is basically the same as the one we introduced in Sect. 2.2. Actually, the major differences emerge in the signature of the definition. Instead of returning a fold, we get a representation of a fold. Clearly, the representation type must be supplied as a type parameter, where we also demand instances of `Symantics` and `CoupleModel` for it.

We can carry out the same exercise with the organization example. Again, we introduce abstract representations of domain optics, which are left unimplemented, and we put them together in an Scala trait.

```
trait OrgModel[Repr[_]] {
  def departments: Repr[Fold[Org, Department]]
  def dpt: Repr[Getter[Department, String]]
  def employees: Repr[Fold[Department, Employee]]
  def tasks: Repr[Fold[Employee, Task]]
  def tsk: Repr[Getter[Task, String]]
```

```
trait RGetters extends Getters[λ[x => x]] {
  def comp_gt[S, A, B](u: Getter[S, A], d: Getter[A, B]) = u ▹ d
  def fork_gt[S, A, B](l: Getter[S, A], r: Getter[S, B]) = l ▵ r
  def id_gt[S] = Category[Getter].id
  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 RAffineFolds extends AffineFolds[λ[x => x]] {
  def id_af[S] = Category[AffineFold].id
  def comp_af[S, A, B](u: AffineFold[S, A], d: AffineFold[A, B]) = u ▹ d
  def filtered[S](p: Getter[S, Boolean]) = AffineFold.filtered(p)
  def as_af[S, A](gt: Getter[S, A]) = gt.asAffineFold
}

trait RFolds extends Folds[λ[x => x]] {
  def id_fl[S] = Category[Fold].id
  def comp_fl[S, A, B](u: Fold[S, A], d: Fold[A, B]) = u ▹ d
  def nonEmpty[S, A](fl: Fold[S, A]) = fl.nonEmpty
  def as_fl[S, A](afl: AffineFold[S, A]) = afl.asFold
}

implicit object R extends Symantics[λ[x => x]]
  with RGetters with RAffineFolds with RFolds
```

Fig. 2. Standard Semantics.

```
}
```

Now, we can provide the analogous for `expertise` as well.

```
def expertise[Repr[_] : Symantics : OrgModel](u Task): Repr[Fold[Org, String]] =
  departments ▹ filtered(employees.all(tasks.elem(u))) ▹ dpt
```

At this point, we have defined generic queries which are not coupled to any particular querying infrastructure. In the rest of the paper, we will show how to reuse such queries for generating in-memory, XQuery and SQL queries.

3.3 Standard Semantics

In defining a new language, it is common practice to start with its syntax and type system, and then proceeds to define its semantics. In our case, we have proceeded in reverse: we started with the intended semantics (optics) and created an abstract syntax and type system which mimic its very structure. So, the only thing to say in this section is how to formalize the connection between the syntax and type system of Stateless and concrete optics, its intended semantics.

In the finally tagless style, the interpretations of the language are given by instances of the `Symantics` type class, and the standard or denotational interpretation is no exception. Moreover, since standard semantic domains are reused at the syntactic level, the denotational semantics is trivially implemented: just use the identity type-level function `λ[x => x]`[7] for the `Repr` parameter, and simply map each operator into its semantic counterpart. We find the interpreter that supplies the standard semantics of Stateless in Fig. 2. Particularly, it is represented by the singleton object `R`, which is a common name for meta-circular interpreters.

[7]We need the *kind-projector* plugin to enable this special syntax for type-level lambdas in Scala.

In order to be able to modularize the sample queries based on concrete optics from section 2.2 (by composing their generic versions and the standard semantics), we need also to instantiate the domain models. As shown in the following listing, we proceed in an analogous way as we did for the instance of the `Symantics` type class, i.e. we select the identity lambda as representation and we reuse the concrete domain optics from Sect. 2.2 to implement the abstract definitions:

```
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
}
```

Now, we can reimplement the sample queries as follows:

```
val differencesR: Fold[Couples, (String, Int)] = differences[λ[x => x]](R, CoupleExampleR)
```

and

```
def expertiseR(u: String): Fold[Org, String] = expertise(u)
```

Note that the required representation and interpretations are inferred implicitly in the second expression.

So far, we have introduced optics and associated patterns and we have generalized them in Stateless. By doing so, we are able to produce generic queries, that we have specialized to concrete optics again. This is not a big deal, but it illustrates the recurrent pattern that we will be using in the next sections.

4 XQUERY

Previously, we have seen that optics serve us to manipulate immutable data structures in a modular and elegant way. However, the state of real applications is mostly handled through databases, web services, etc. This section shows that we can reuse Stateless queries to approach other underlying infrastructures. Particularly, we will translate `differences` and `expertise` into XQuery expressions to query XML documents.

Firstly, we will manually adapt the original queries and their corresponding models into the XML/XQuery setting [29] in an idiomatic way. Secondly, we will introduce the non standard semantics that we could use to translate Stateless queries into XQuery expressions automatically, where we need to make several assumptions about the adapted models.

4.1 XML/XQuery Background

There are different ways of encoding the state of the couple example as an XML document. Fig. 3 shows a possible way of doing so. It contains a root tag `xml` where couples hang from, as `couple` tags, which in turn contain tags for the woman (`her`) and man (`him`) conforming the couple. Finally, `name` and `age` are simple tags that contain primitive values.

Usually, an XML document is accompanied by an XSD schema, which turns out to be essential to validate the information that we place in the document. You can find the schema associated to the couple document in A.1. Among other things, it prevents us from defining people without a `name` tag, placing non numerical values as `age` values and defining several `her` tags inside a couple. Later, we will see that it is important to take the schema into account while implementing queries.

Now, we would like to produce an XQuery expression, analogous to `differences`. It should be able to collect the name and age difference of all woman who are older than their mates. Since we do not want to calculate a single value for this query, like a number or a boolean, the results should

```
<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>
```

Fig. 3. Couples deployed as XML.

be presented as a sequence of nodes, i.e. the output is also an XML tree. This could represent the output of the query we are looking for

```
<xml>
    <tuple>
        <fst><name>Alex</name></fst>
        <snd><diff>5</diff></snd>
    </tuple>
    <tuple>
        <fst><name>Cora</name></fst>
        <snd><diff>2</diff></snd>
    </tuple>
</xml>
```

Note that we pair values by means of a contrived `tuple` tag, which contains `fst` and `snd` projection tags where data is finally contained. Once we know what is the output that we want to produce, we show you the XQuery expression that we could use to generate it.

```
/xml/couple[her/age > him/age]/<tuple>
                                  <fst>{her/name}</fst>
                                  <snd><diff>{her/age - him/age}</diff></snd>
                                </tuple>
```

We describe its components in the following paragraphs.

One of the most fundamental queries is `/`, which grants access to the document node, which we can see as the entry point in the document. Since XML documents are essentially nested data structures, XQuery provides a short syntax to access nested tags. For example `/xml/couple` selects all tags `couple` which are hanging from a tag `xml` which in turn should be accessible from the document node.

XQuery does provide filters to enrich queries, which are placed inside square brackets. For example, `[her/age > him/age]` is a filter that we apply over `/xml/couple` to discard the couples where her age is not bigger than him age. Notice that the operator `>` is able to extract the inner value of these tags and interpret them as numbers. Notice that this is perfectly safe, if we take into account the XML schema.

XQuery supports XML interpolation to enrich its results. It serves us to provide the surrounding structure that we need to put pairs of values together. It is worth mentioning that this is the only feature from XQuery which is not also available from XPath among the ones we use in this work.

Now, we could adapt the organization example, along with the `expertise` query. Fig. 4 shows the XML document where we adapt the information from the original example. Again, this document is valid according to the schema that we have placed in A.2.

```
<xml>
    <department>
        <dpt>Product</dpt>
        <employee><emp>Alex</emp><task>build</task></employee>
        <employee><emp>Bert</emp><task>build</task></employee>
    </department>
    <department>
        <dpt>Quality</dpt>
    </department>
    <department>
        <dpt>Research</dpt>
        <employee><emp>Cora</emp><task>abstract</task><task>build</task><task>design</task></employee>
        <employee><emp>Drew</emp><task>abstract</task><task>design</task></employee>
        <employee><emp>Edna</emp><task>abstract</task><task>call</task><task>design</task></employee>
    </department>
    <department>
        <dpt>Sales</dpt>
        <employee><emp>Fred</emp><task>call</task></employee>
    </department>
</xml>
```

Fig. 4. Organization deployed as XML.

As you already know, `expertise("abstract")` returns the name of the departments where all the employees are able to *abstract*. Again, we need to return a node sequence, since we could find many departments matching the criteria. Thereby, the output that the XQuery should produce might be

```
<xml>
    <dpt>Quality</dpt>
    <dpt>Research</dpt>
</xml>
```

Producing such a query, analogous to `expertise`, is not straightforward, since we do not know of any standard XQuery method to check if all elements which are hanging from certain context do hold a predicate. Fortunately, we have mechanisms to know if a query produces any result and to negate booleans, so we can implement the desired behaviour in terms of them.

```
/xml/department[not(employee[not(task[tsk = "abstract"])])]/dpt
```

The query produces the expected results, but we find it difficult to read, due primarily to the combination of filters and negations. We describe the new XQuery elements which appear here in the next lines.

Firstly, there are several invocations to a `not` function. This is just the negation function that we could find in many programming languages, but it deploys extra functionality beyond negating booleans. In fact, it also produces `true` if the argument corresponds to a non-empty sequence of elements, and `false` if the argument corresponds to an empty one.

Secondly, we find a new operator `=`, which corresponds to equality. The first operand in the equality is a tag so, its value is extracted. The second one is a string literal. Beyond strings, XQuery provides literals for other basic types, like numbers or booleans.

4.1.1 Scala embedding of XQuery. Before presenting the non standard semantics, we find it important to show you the algebraic data type that we use to represent XQuery expressions in Scala, since we will use it in the `Symantics` instantiation. You can find such data type in Fig. 5. Clearly, this is just a simplification of XQuery grammar, which contains exclusively the language features that we will use along this section.

We introduce now the embedded version of the previous queries. We adapt `differences` as follows

```
val differencesXQuery: XQuery =
  Seq(Document, Seq(Name("xml"), Seq(Name("couple"), Seq(
    Filter(Oper(">", Seq(Name("her"), Name("age")), Seq(Name("him"), Name("age")))),
```

```
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 Tuple(fst: XQuery, snd: XQuery) extends XQuery
```

Fig. 5. XQuery algebraic data type.

```
  Tuple(
    Seq(Name("her"), Name("name")),
    Oper("-", Seq(Name("her"), Name("age")), Seq(Name("him"), Name("age")))))))
```

and we also embed `expertise` in the next definition

```
def expertiseXQuery(u: String): XQuery =
  Seq(Document, Seq(Name("xml"), Seq(Name("department"), Seq(
    Filter(Func("not", Seq(Name("employee"),
      Filter(Func("not", Seq(Name("task"),
        Filter(Oper("=", Name("tsk"), PString(u)))))))),
    Name("dpt")))))
```

We briefly introduce the data type components in the next paragraphs.

As we can see, both expressions start with `Seq`. This value is widespread in both expressions, since it is the glue that puts two subexpressions together. In fact, it has two major purposes. Firstly, we can see it as the `/` in nested access, like in `p/q`. Secondly, we can see it as a delimiter between context and filter, like in `p[q]`. Clearly, we will find nested `Seq` occurrences when we want to connect more than two subexpressions.

There are several components that should be straightforward now. `Document` corresponds to the document node. `Name` represents the selection of a tag whose name is taken as a parameter. Functions and operators are represented with `Func` and `Oper`, respectively. They both take the name of the abstraction as the first parameter, and operands follow. There are also expressions to reify literal values, like `PString` for the particular case of strings.

For the sake of simplicity, we supply a `Tuple` value, which we use as a shortcut to represent the interpolation of the `<tuple/>` tag. It takes two subexpressions as parameters, that corresponds to the values that should be placed in the `<fst/>` and `<snd/>` projection tags, respectively.

Finally, we find a new element that we have deliberately ignored so far, since it was not introduced in the queries. It is the `Self` axis that refers to the current context. In XQuery, it is represented as a `.` (dot). This self notion is redundant under nested access. For example `./couple/./her/.` is equivalent to `couple/her`. We will need this self notion later on.

4.2 XQuery Non-standard Semantics

We come back to our objective of interpreting Stateless queries into XQuery expressions. If we recall `differences` and `expertise`, they are parameterized over the type of representation. Since we aim at generating an XQuery expression, we should consider using its embedded form as representation. However, `XQuery` is untyped, so we need to adapt it into a type constructor. We can achieve this task by using `λ[x => XQuery]` instead –which is just a convenient way of encoding the *Const* functor.

Once we have identified a candidate representation, we show the whole expression that adapts a generic Stateless query into an XQuery expression. For `differences` we have

```
val differencesXQuery: XQuery = differences[λ[x => XQuery]].setRoot
```

and for `expertise` we get

```
def expertiseXQuery(u: String): XQuery = expertise[λ[x => XQuery]](u).setRoot
```

Instead of ad hoc monomorphic definitions, these expressions produce the corresponding XQuery expressions in a modular way, by reusing Stateless queries. The rest of the section aims at filling in the missing pieces from the previous definitions. Firstly, we will introduce model instances for the new representation. Secondly, we will present its symantics instance. Both are necessary to meet the constraints of Stateless queries. Finally, we will show what is the precise purpose of `setRoot`, where the final result is produced.

4.2.1 Model translation. Before instantiating the corresponding typeclasses, there are several assumptions about the adaptation of the models that we would like to make, where we basically adopt the same convention that we saw in the background.

Firstly, we will assume that all the information is hanging from an `xml` tag, which acts as the root of the XML document. Secondly, we will assume that every optic corresponds to an XML tag, where the optic kind determines the tag cardinality. Finally, optics pointing at base types are adapted as *simple type* elements containing a value with the corresponding basic type; optics pointing at entities are adapted as elements with *complex type*, since they nest other elements. In this sense, we refer to *entity* as a domain-specific type, like `Couples` or `Employee`. Each of the previous restrictions is assumed in the XSD schemas.

Now, we have all the ingredients that we need to provide the instances of `XQuery` for generic models.

```
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")
}
```

As we said before, optics correspond to XML tags, so we represent them as mere tag selection using `Name`. Optic names are good candidates as tag names. However, we need to adjust the plural names of folds into singular ones, like in `Name("couple")`, since this information is deployed in individual tags. The instance for the organization model should be straightforward now and does not add any value, so we do not show it.

4.2.2 Symantics instance. As a final step towards the interpretation of Stateless expressions, we need to provide an instance for `Symantics`. The whole non-standard semantics are collected in Fig. 6. In the next paragraphs, we will introduce the different methods, which are organized by the kind of the involved optics, as usual.

We start with the categorical definitions for getter in `XQueryGetters`. Firstly, $\texttt{comp}_{gt}$ is translated as a `Seq` where the composing expressions are tied together. For $\texttt{fork}_{gt}$ we use `Tuple`, which interpolates the input expressions with the tuple structure that we had determined to pair values. Finally, $\texttt{id}_{gt}$ is interpreted as a `Self` reference, which is neutral under composition. Now, we move on to standard getter constructions, beginning with `like`. Since it produces constant optics, whose focus does not depend on the surrounding source, we decide to map them into XQuery literals. Next, we can see that `not` is interpreted as the function `not` and `eq`, `gt` and `sub` are interpreted as the corresponding operations.

Regarding `XQueryAffineFolds`, we find `filtered`. Since we have a filtering mechanism available in XQuery, we simply interpret this primitive into `Filter`, passing the semantics of the predicate

```
trait XQueryGetters extends Getters[λ[x => XQuery]] {
  def id_gt[S] = Self
  def comp_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 eq[S, A: Base](x: XQuery, y: XQuery) = Oper("=", x, y)
  def gt[S](x: XQuery, y: XQuery) = Oper(">", x, y)
  def sub[S](x: XQuery, y: XQuery) = Oper("-", x, y)
}

trait XQueryAffineFolds extends AffineFolds[λ[x => XQuery]] {
  def id_af[S] = Self
  def comp_af[S, A, B](u: XQuery, d: XQuery) = Seq(u, d)
  def filtered[S](p: XQuery) = Filter(p)
  def as_af1[S, A](gt: XQuery) = gt
}

trait XQueryFolds extends Folds[λ[x => XQuery]] {
  def id_fl[S] = Self
  def comp_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
}

implicit object XQuerySymantics extends Symantics[λ[x => XQuery]]
  with XQueryGetters with XQueryAffineFolds with XQueryFolds
```

Fig. 6. XQuery non-standard semantics.

getter as an argument to it. The categorical definitions are identical to the ones for getters. The casting from getters to affine folds becomes the identity function. This situation also occurs in fold combinators, which evidences that we do not make a difference between semantic domains in the interpretation. In fact, if we understand `XQuery` as a representation of a fold, it is natural that we can also use it as a representation of a getter or an affine fold.

Lastly, we present the fold related method `nonEmpty`. In this particular case, we need to adapt any fold into a getter pointing at a boolean. Luckily, XQuery provides a function `exists` which turns XQuery expressions into booleans. It does so by checking that the result that the query produces is not empty. You might have noticed that `exists` was not even mentioned in the background section. In fact, it was not necessary, since the `not` function does the trick, by turning an expression denoting a sequence into a boolean. Particularly, `not(exists(sq))`, where `sq` denotes a sequence of elements, is equivalent to `not(sq)`. However, while interpreting, we do not know if this expression will be consumed by a function expecting a boolean or not, so we need to invoke `exists` explicitly.

4.2.3 Running examples. Once we have defined instances for models and symantics, we should be able to recover XQuery expressions. However, as we said before, we need to invoke `setRoot` to get the final query. This is just a consequence of one of the assumptions that we made when adopting XML, where we stated that an `<xml/>` root tag was necessary. Indeed, this method just prepends such tag and provides access to the document node.

```
implicit class XQueryOps(xq: XQuery) {
  def setRoot: XQuery = Seq(Document, Seq(Name("xml"), xq))
```

```
}
```

Thereby, `differencesXQuery` provides the following XQuery expression, which we present in its textual version, to facilitate its understanding.

```
/xml/couple[her/age > him/age]/<tuple>
                                  <fst>{her/name}</fst>
                                  <snd>{her/age - him/age}</snd>
                               </tuple>
```

There is a slight difference if we compare it with the original one: the second tag in the output is not surrounded by a `diff` tag, since optics do not provide a standard way of supplying aliases for expressions. The rest of the query remains the same.

We also supply the output provided by `expertiseXQuery("abstract")`

```
/xml/department[not(exists(employee[not(exists(task/tsk[. = "abstract"]))]))]/dpt
```

We can see a self reference (dot) when comparing the task with the literal `"abstract"`, consequence of the particular implementation of `elem` that we presented in Sect. 3.1, which uses the identity getter to refer to the predicate parameter while invoking `any`. Besides, we find redundant invocations to `exists`[8]. If we ignore these minor differences, the query is essentially the same as the one we presented at the beginning of this section, and therefore produces the very same output.

As final notes, we must say that ignoring `setRoot` leads to relative queries, i.e. queries that do not start with `/` and which are relative to the current context. Those queries are valid XQuery expressions, but they will not produce any result if we run them against the XML document which contains the whole hierarchy. The good thing is that we could easily compose such relative queries with the ones generated by external models to produce queries over more complex XML files.

5 SQL

This section aims at translating Stateless optic expressions into SQL statements. Firstly, we will manually adapt the couple and organization examples into the SQL setting, to better understand the kind of queries that we want to produce. Then, we will present the SQL non-standard semantics and the assumptions that we need to make in order to generate analogous queries from Stateless expressions.

5.1 SQL Background

As opposed to XML, relational databases are organized as flat data. More specifically, the information is structured as tables, which in turn contain columns, where data is stored. We propose the following tables to adapt the couple example model that we introduced in 2.2.1:

```
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)
);
```

As you can see, case classes are adapted as tables and their attributes are adapted as columns. Once again, as we saw in the XQuery interpretation, it is necessary to distinguish between attributes

[8] These invocations could be removed from the resulting query by means of annotations, as in [25], but we wanted to keep the interpreter compositional, to make it simpler.

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`

Fig. 7. Initial state for couple example tables.

pointing at base types and attributes pointing at other entities. In fact, attributes that refer to entities require keys to establish the precise connections between the adapted tables. Particularly, we need them to reassemble the nested structure where we have couples pointing at people. We assume Fig. 7 as the initial state for these tables, where the columns in `Couple` are clearly pointing at names in `Person`.

As we have already seen, the adaptation of `differences` in the XML setting produced XML as output. Here, we are dealing with SQL tables, where the output of a statement is a table itself. Thereby, we would expect the following result after executing the adaptation of `differences` in this new setting.

name	diff
Alex	5
Cora	2

Fortunately, we can use **SELECT** statements to select data from a database. In particular, we could produce the previous output by adapting `differences` into this kind of statement as follows.

```
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;
```

As you can see, this query is separated in three major sections. Firstly, we find the **SELECT** clause itself, where we indicate the columns that we are interested in: the name of the woman and the age difference. Secondly, **FROM** builds the raw table that the other parts use to gather information from: it introduces variables to refer to a woman `w` and a man `m` conforming a couple. Finally, **WHERE** introduces filters that are applied over the generated table to discard the rows that do not match the criteria: those where the age of the woman is not greater than the age of the man.

We adapt now the organization example. First of all we create tables for departments, employees and tasks.

```
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 (
```

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`

Fig. 8. Initial state for org example tables.

```
  tsk varchar(255) NOT NULL,
  emp varchar(255) NOT NULL,
  FOREIGN KEY (emp) REFERENCES Employee(emp)
);
```

All components in the previous statements should be familiar at this point, but there is an important change in the way we deploy keys. If you recall from the couple example, getters pointing at entities were mapped into a column containing a foreign key. However, the organization example contains multivalued attributes, like `employees` or `tasks`, that should not be adapted as a single column. For this situation, we adopt the one-to-many relationship which is widespread in the SQL folklore[9]. We assume that these tables have been populated with the data in Fig. 8.

As we have already seen, *Quality* and *Research* are the departments where all employees are able to abstract, so the adaptation of `expertise("abstract")` should produce the following table when executed.

dpt
Quality
Research

Finally, we present the adapted query that we could use to generate it.

```
SELECT d.dpt
FROM Department AS d
WHERE NOT(EXISTS(SELECT e.*
                 FROM Employee AS e
                 WHERE NOT(EXISTS(SELECT t.*
                                  FROM Task AS t
                                  WHERE (t.tsk = "abstract") AND (e.emp = t.emp)))
                       AND (d.dpt = e.dpt)));
```

Reading this query is not trivial at all. Fortunately, it shares the same pattern than the adapted version of `expertise` for XQuery. In fact, **`EXISTS`** is a function that returns true as long as the nested statement produces non-empty results. If we combine it with **`NOT`** to negate predicates, we can check if all rows hold a condition. Beyond the noise generated by this pattern, there are additional filters which manifest relations between nested and outer statements that introduce even more complexity in the query.

[9] https://stackoverflow.com/a/7296873/1263978h

5.2 SQL Non-standard Semantics

As we have just seen, a SQL select statement exhibits a remarkable separation of aspects, where selection and filtering, though sharing syntax, belong to different query clauses. This separation requires a unifying mechanism to refer to the very same thing from the different parts. This is solved by means of variables which are declared in the **FROM** clause and are accessible from both **SELECT** and **WHERE** scopes.

This way of representing queries contrasts with its optic counterpart. In optics, the aspects of selection and filtering are entangled, in the sense that they can appear anywhere in the expression. In this scenario, it is the context where two optics appear the one that determines whether they are pointing at the same thing, so no variables are needed. For example, consider the expression

```
couples ▹ filtered (her ▹ age < 50) ▹ her ▹ name
```

which is just a less direct way of implementing `under50` from Sect. 1, where we find two occurrences of `her`. Despite having one of them surrounded by `filtered`, we can see that they are pointing at the very same structure. As a final remark, it might be worth mentioning that `filtered` expressions have their own identity, unlike **WHERE** clauses.

As a consequence, if we aim at translating Stateless expressions into SQL queries, we need to contemplate the aforementioned disagreement. For this mission, we will use an intermediate data structure where the separation of concerns is made explicit, that we will refer to as `TripletFun`. We will provide the details around this data type further on. For the time being, we can conform the whole expression that turns a Stateless expression into a SQL statement. For `differences` we get

```
val differencesSQL: Error \/ SQL =
  differences[λ[x => TripletFun]].genSQL(Map("Person" -> "name"))
```

and `expertise` is adapted as follows

```
def expertiseSQL(u: String): Error \/ SQL =
  expertise[λ[x => TripletFun]].genSQL(Map("Department" -> "dpt",
                                           "Employee"   -> "emp")
```

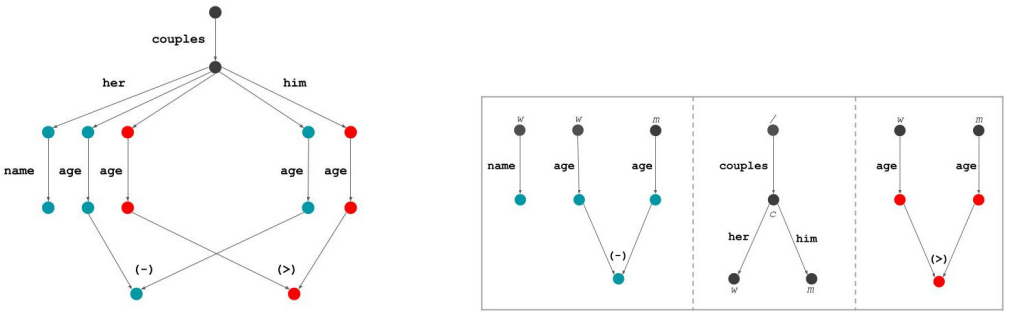
Again, we embrace the constant type lambda to adopt `TripletFun` as representation. Recall that we need to instantiate the corresponding model and symantics for such type constructor if we want to invoke Stateless queries. Then we find an invocation to `genSQL`, which is responsible for turning the `TripletFun` into a SQL statement. Notice that it takes a key-value map as argument. It corresponds with the relation of primary keys, essential information that is not contemplated in the optic model. Note as well that generating SQL could lead to errors, as suggested by the signature. We will come back to these aspects later.

The next sections introduce model and symantics instances, as well as the details of SQL generation. Prior to that, we will describe the details of the intermediate structure `TripletFun`.

5.2.1 Intermediate structure. The main objective of `TripletFun` is to provide a structure where the selection and filtering aspects from an optic expression are decoupled. We present an informal view of the idea in Fig. 9.

On the left hand side we can see the original `differences` expression represented as a kind of directed graph. Notice that the edges of this graph are optics, so a path corresponds to a sequence of optic composition. In the situation where a node is pointed by two nodes, we are representing operations connecting operands. Blue dots belong to paths that are involved in the selection aspect, while red dots belong to paths that are involved in the filtering aspect. Finally, black dots are the ones that belong to paths which are shared by both groups.

On the right hand side of the figure we can see the decoupled representation associated to the very same expression, where different aspects have been segregated in a triplet. The middle



(a) Raw expression. (b) Decoupled data structure.

Fig. 9. From original expression to triplet.

component in the triplet is a tree where we place all those paths that are potentially reusable. Indeed, paths are identified by a textual label to make it easier to refer to them. We have determined that the tree should contain exclusively those paths which point at entities from the model, i.e. the only optics that we could find as nodes in the tree should be the ones that are pointing at an entity. Once we have introduced the path tree, it is easy to describe the rest of components in the triplet. The first one reuses the paths `w` and `m` to pick her name and the age difference. In the third component we find the filtering aspects, which are completely isolated from the selection ones. This path reuses the same paths, this time to filter out those women whose age is not greater than the one from her mate.

Now, we show you the way we have encoded this intermediate data structure in Scala.

```
type Triplet = (List[TExpr], VarTree, Set[TExpr])
```

As you can see, it deploys three components, which correspond to the selection, variables and filtering aspects, respectively. We start with `VarTree`

```
type VarTree = ITree[OpticType, (String, Boolean)]
```

which is a kind of rose tree where subtrees are indexed by the basic information associated to an optic: kind, name, involved types, etc. Nodes are labelled with the name of the path and by a boolean value which is neccessary to deal with nested queries, as we will see later. The expressions that we find in the selection and filtering components of the triplet are encoded as collections containing elements with type `TExpr`. In essence, this data type represents any optic expression, but it has been extended with the ability to point at paths in the variable tree and it restricts vertical composition to mere projections. Notice that the first component is a list of expressions, where the expressions conforming the selection could be repeated. On the other hand, the third component is a set of expressions, since repeated filters would be redundant.

At this point, we could consider using `Triplet` as a final representation, having each Stateless primitive generating the corresponding triplet. However, composing the different triplets generated by optic subexpressions turns out to be a clumsy task. Instead, we would like to use a representation with better compositional guarantees. In this sense, we find it convenient to understand the interpretation as a triplet transformation, where each subexpression describes the precise transformation that it should achieve over the surrounding triplet. That is how we get to `TripletFun`.

```
type TripletFun = Triplet => Triplet
```

We illustrate the idea behind this function in Fig. 10, which shows the evolution of the triplet for the `differences` query, starting from the empty triplet, where the changes introduced by each

subexpression are emphasized. As expected, the last triplet in the chain corresponds to the structure that we presented in the right hand side of Fig. 9. We will detail these steps along the next sections.

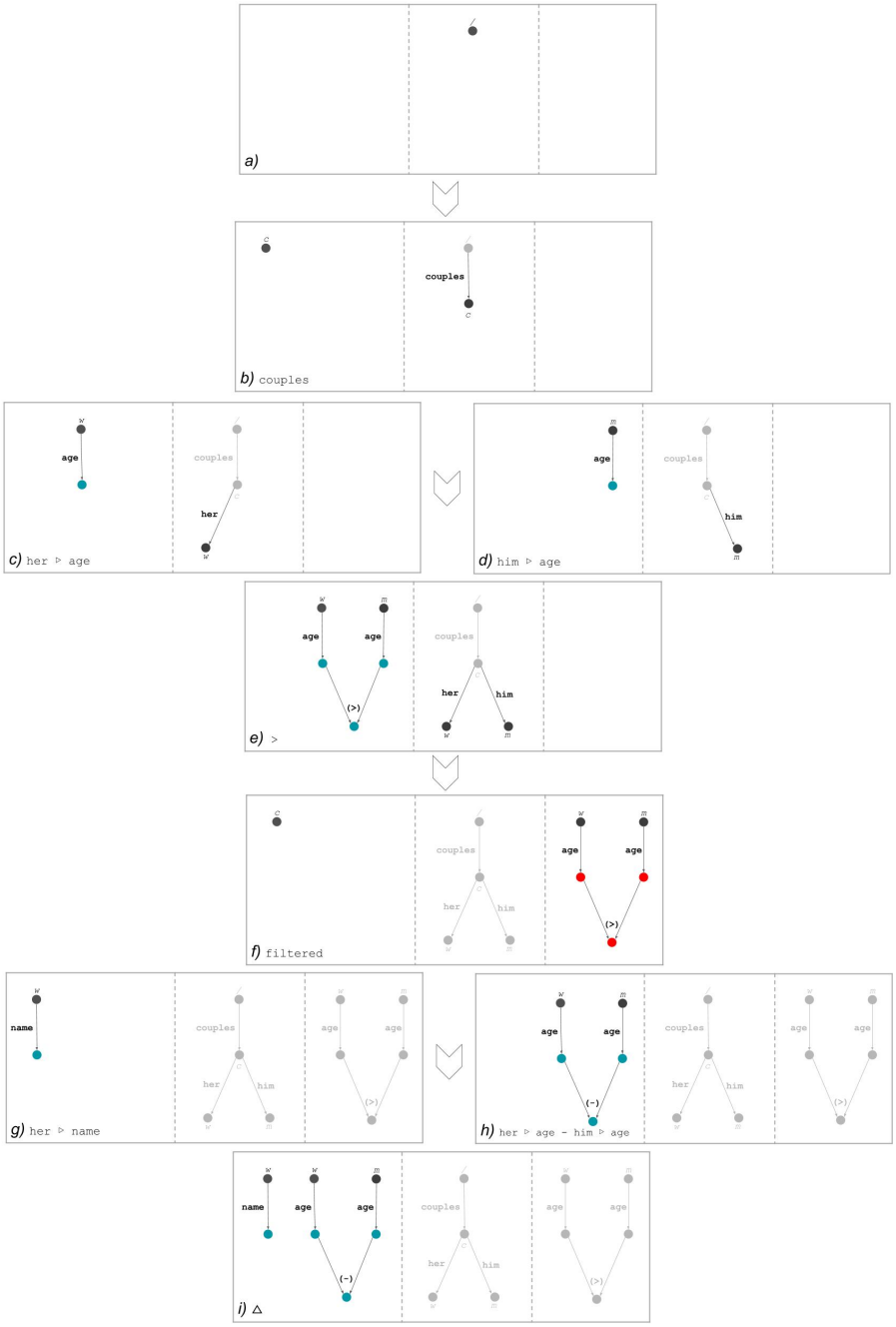


Fig. 10. Triplet evolution for `differences`

5.2.2 Model translation. If we aim at instantiating models, we need to know how to adapt domain optics into triplet functions. In other words, we need to describe what is the precise transformation that an optic carries out over a triplet. Previously, we could see that optics pointing at entities are collected in the variable tree. However, optics pointing at base types were selected as projections of fields acting on variables. This aspect seems relevant to identify the transformations that we are looking for.

As we said before, Fig. 10 shows the triplet evolution for `differences`, starting from the empty triplet in *a)*. The first step *b)* is the result that we should get once we apply the transformation produced by `couples`. This optic is pointing at the entity `Couple`, and therefore it should be included as a new entry in the variable tree. Besides, we include the new variable in the selection component. Since no filter is involved here, we leave the third component empty.

In *c)*, we show the changes that are produced over *b)* once we apply the transformation associated to `her ▸ age`. Again, `her` is an optic pointing at an entity, so it must be placed in the variable tree. However, we do not place it in the root. Instead, we graft it from the variable introduced for `couples`. This decision is determined by inspecting the selection component in *b)*. If it is empty, we graft the new variable in the root of the tree. Otherwise, we graft it in the path were the variable is pointing at. Anyway, the new variable is introduced in the selection component.

The state in *c)* is also influenced by the optic `age`, which points at a base type. Thereby, it does not produce any transformation on the variable tree, although it updates the selection component. Particularly, it expects to find a variable in the transforming state and it extends it with a projection to the involved field which is pointed by the optic.

Now, we show you how to encode the previous transformation in the model instances. We use `entity` and `base` for this mission, which are responsible for producing the corresponding `TripletFun` implementations. They take the optic information as input. In addition, `entity` also takes the path name as second argument[10]. We show the instance of the couple model for the adopted representation.

```
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"))
}
```

Note that `entity` is used for optics pointing at entities and `base` is applied otherwise. It might be worth mentioning that `GetterType` and `FoldType` are just `OpticType` cases, which collect basic information around the involved optic. We do not show the organization model instance since it should be straightforward.

As final remark, we must say that both `entity` and `base` are partial, where a single path in the selection component of the transforming triplet is assumed. This is granted by construction, as long as we restrict models to contain optics whose source types are not base types.

5.2.3 Symantics instance. This section introduces the triplet transformations which are associated to the primitives in `Symantics`. You can find the particular instance in Fig. 11. As usual, `TripletFunSymantics` is presented in a modular way, where submodules are organized by optic kind.

Getters. Firstly, we present the interpretation of $\texttt{comp}_{gt}$. If the current representation consists of a triplet function, it is natural to understand this primitive as mere function composition, where transformations are chained. Consequently, $\texttt{id}_{gt}$ is implemented in terms of `identity`, meaning no

[10] We will assume that such names are unique for simplicity.

transformation at all. The horizontal composition introduced by $\texttt{fork}_{gt}$ is slightly more complicated. We introduce a new combinator `merge3With`, which is responsible for generating the composed `TripletFun`. The result that it produces applies the transforming triplet to both `l` and `r` in parallel, so we get a pair of independent triplets. The combinator takes three merging functions to fusion the pair of triplets into a unique one, componentwise. In this sense, it is worth mentioning that `lmerge` joins the variable trees, so that identical paths are merged into the same one. The other two functions, the ones that merge the selection and filtering components, are just concatenation of collections.

We can see an example of the transformation introduced by $\texttt{fork}_{gt}$ in Fig. 10, where *g)* and *h)* correspond to the pair of independent triplets which are generated for the left and right operands of the fork. Particularly, *g)* selects her name and *h)* selects the age difference. Both states are merged, deriving in *i)*. Merging the very same variable trees and filters produces no effect, but the selection component is updated by putting the forking selections together.

Next, we find `like` and `not` as examples of unary standard combinators, which just update the selection component of the triplet[11]. The first of them ignores the previous selection and replaces it with the constant value. The second one transforms the triplet by wrapping the selection with the operator, where an error is raised if we find multiple values in the surrounding selection, which should never happen by construction. Finally, `gt` (`>`) interpretation is basically the same as the one we saw for horizontal composition, but the selection components are fused into the corresponding operation, instead of being packaged in a list. In fact, you can see an example of the transformation it produces in the states labelled as *c)*, *d)* and *e)*, where her age and him age are fused by means of `>`.

Affine Folds. As in the XQuery interpretation, categorical methods are exactly the same as the ones we presented for getters, and castings become the identity function for analogous reasons. The same idea will apply to folds.

The primitive `filtered` is the only one that introduces new elements in the filtering component of the triplet. Clearly, an expression does not know that it will conform the predicate of a filter when it is being constructed. Thereby it will assume that it is selecting a boolean element as output. As a consequence, `filtered` must be responsible for reorganizing the information that is supplied in the interpretation of the predicate. Since `filtered` does not alter the current selection, the selection component passes as is. However, it might introduce updates in the variable tree and it definitely adds new elements in the filtering component. For this task, we need to supply the transforming triplet to the predicate semantics[12]. We choose the transformed tree as variable tree, since new variables might have been added by the predicate expression. Besides, we append the contents of the selection component to the standing filtering component.

Back to Fig. 10, we find an example of the transformation that is introduced by `filtered` in the states *e)* and *f)*. In essence, *e)* is the triplet which is produced by the predicate. As we can see, its selection is moved to the filtering component in *f)*, we pick the variable tree from the predicate and we restore the selection from *b)* as is —which is the state that we had before applying the predicate transformations.

Folds. Finally, we present the interpretation of `nonEmpty`, as member of the primitives associated to folds. We find it convenient to keep this primitive in its own scope, since it is usual to have it interpreted as a subquery, as we will see in the SQL generation. Thereby, it does not transform the variable tree or the filters from the original triplet. Nevertheless, it creates its own scope living in the selection component, which is just a triplet wrapped in a `NonEmpty` expression. As you can see,

[11] Interestingly, we use the standard lens `first` for this task.

[12] We supply an empty filter component and we ignore it in the resulting triplet, since getters are not able to extend it.

```
trait TripletFunGetters extends Symantics[λ[x => TripletFun]] {

  def id_gt[S] = identity

  def comp_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 gt[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 TripletFunAffineFolds extends Symantics[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 TripletFunFolds extends Symantics[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)
  }
  ...
}

implicit object TripletFunSymantics extends Symantics[λ[x => TripletFun]]
  with TripletFunGetters with TripletFunAffineFolds with TripletFunFolds
```

Fig. 11. Triplet non-standard semantics.

this triplet is generated by an invocation to `fl`, that corresponds to the semantics of the input fold. Interestingly, the variable tree that we pass as argument marks all variables as external, to inform the subquery that they were introduced in an outer scope, introducing therefore a distinction between local variables introduced by the nested fold and external variables.

5.2.4 Generating SQL. Once we have generated the triplet transformation, we provide a brief tour on the generation of the final SQL statement. Fortunately, the separation of aspects in the intermediate structure represents the bulk of the work. However, there are still several subtleties that we need to take into account. We present them in the following paragraphs.

First of all, we make several assumptions over the adaptation of the model to SQL, which are essentially the ones we adopted in the background section. Particularly, entities are translated into tables and base types are understood as table column types. In this sense, getters and affine folds pointing at entities should be adapted as foreign keys. If they are pointing at base types, they represent columns. On its part, folds pointing at entities assume that the target entity is pointing at them by means of a foreign key, as we introduced in the background section. We do not support folds pointing at base types, since they would lead us to a weird situation. Concretely, it is a bad practice to keep several values in a column entry. Luckily, we can circumvent this situation by introducing an entity wrapper, as we did with `Task` in the organization example.

As we saw at the beginning of this section, the SQL statement is generated by `genSQL`, which takes the relation of primary keys as parameter. The first mission of this method is to generate the triplet, by supplying the empty triplet as initial state. Now, we can interpret each component. Figure 12 shows the particular clauses that it generates for the triplet associated to `differences` that we presented in Fig. 9. As expected, the selection and filtering components correspond to the **`SELECT`** and **`WHERE`** clauses of the statement. The translation of the expressions in both components is the same, but the delimiter is different. In this sense, we separate the components of the selection component by means of `','`, but we use **`AND`** to delimit the filtering expressions, since all predicates must hold.

The translation of an expression is trivial. If we find a raw variable `v` we interpret it as `v.*`, where we select the whole value. Projections and operations are adapted into their counterparts in SQL. To adapt the semantics associated to `nonEmpty`, we need to invoke the interpreter recursively and we wrap the generated query as an argument for `EXISTS`. Note that the generator is also responsible for generating the extra filters that we need to connect subqueries with the outer scope, which were introduced in the background section.

Translating the variable tree is not that easy, since the SQL generator needs to take keys into account. Basically, the generator traverses the tree to produce a sequence of **`INNER JOIN`** expressions, which are placed in the **`FROM`** clause. Clearly, these expressions require a condition to state the precise relation between tables. For this task we need to consult the involved tables, information which is encoded in the links of the variable tree, and we need to know the primary key of each table, information which is supplied explicitly to `genSQL`, since it is not contemplated in the optic model. Note that variables marked as external are not included in the generated sequence.

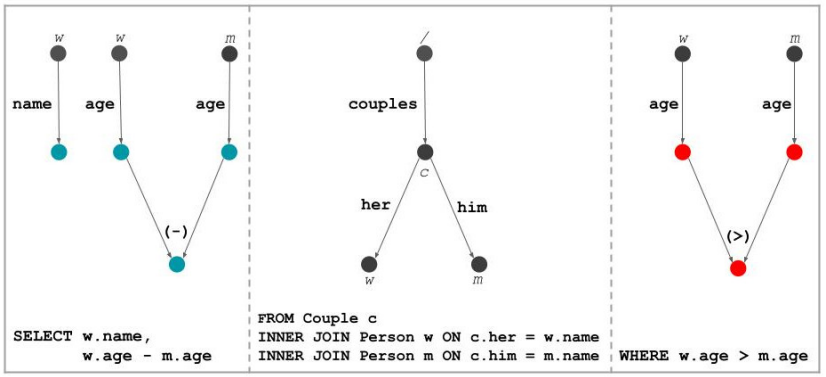


Fig. 12. From triplet to SQL

The queries that the generator produces for `differencesSQL` and `expertiseSQL("abstract")` are exactly the same as the ones we introduced in the background section.

As final remarks, we must say that the generator is partial, for several reasons. First of all, the relation of primary keys could be wrong. Secondly, we could get expressions pointing at non existing variables in the tree, although this situation should never happen if the triplet was generated by the interpreter. Finally, the tree structure that we introduced in the model guarantees that all queries are correct, as long as they start from the root of the model, like queries starting with `couples` or `departments`. However, queries like `her ▸ name` are clearly incomplete, since they need to find their path towards the root. In this sense, we could generate valid SQL statements by using the suitable triplet as initial state instead of the empty one.

6 DISCUSSION

The language of optics. One the most prominent sought-after features of optics is *modularity*, i.e. the capacity of creating optics for compound data structures out of simpler optics for their parts. This is specially emphasized in the framework of profunctor optics [24], where optic composition builds upon plain function composition, and enables straightforward combinations of isos, prisms, lenses, affine traversals, and traversals. The profunctor representation is particularly convenient to implement (and even reveal) the compositional structure of the different varieties of optics, but, in essence, this structure is also enjoyed by concrete optics, Van Laarhoven optics, etc. Modularity is a feature of the *language* of optics, rather than of any particular representation. This paper has shown, albeit for a very restricted subset of optics (getters, affine folds and folds), that this compositional structure of optics can be encoded in the type system of a formal language, that we named *Stateless*. The denotational semantics of this language was given in terms of concrete optics, but any other isomorphic representation, such as profunctor optics, may have served as well.

However, the compositional features of optics only represent part of their specification. The other essential ingredient of their meaning corresponds to the kinds of queries enabled by those data accessors: retrieving, updating, etc. For instance, folds should enable a *getAll* query to retrieve from a given data structure[13] the sequence of elements they point to. Admittedly, the current version of Stateless only formalizes the compositional part of optics, but not their intended queries (i.e. there is no abstract *getAll* signature in the *Folds* type class). Taking into account this non-compositional character of optics is essential as soon as we tackle the extension of Stateless with new varieties of optics, specially the ones that allow us to update the content of data structures[14]. For instance, the major difference between folds and traversals are not in their compositional properties, but in the queries they must support: besides *getAll*, traversals must also support a *putAll* query to replace the content of the elements they point to.

This separation of concerns between declaratively *pointing* to parts of a data structure, and building a variety of *queries* related to those parts, is the cornerstone of optics. In this regard, the LINQ approach based on comprehensions focuses on the query building side, and, commonly, on constructing queries of a simple kind: retrieval queries denoting a multiset (the semantic domain for queries on QUEA [25], T-LINQ [7], NRC [2], etc.)[15]. The optics approach is hence more modular. For instance, a representation of traversals intended for SQL should allow us to generate both a **`SELECT`** and an **`UPDATE`** statements for the queries *getAll* and *putAll*, respectively. We plan to deal

[13] Understanding "data structure" in a general sense, i.e. as an XML document, a relational database, etc.; not only as an immutable data structure

[14] Beyond the small subset of read-only optics that we considered in this paper, there exists a huge catalogue of optics in the folklore that we could use to enrich Stateless (cf. http://oleg.fi/gists/images/optics-hierarchy.svg)

[15] Of course, practical query languages like Quill must handle inserts, updates and deletes, but this is performed with an ad-hoc DSL, which escapes the collection-like interface.

with this extension in future versions of Stateless. We also leave to future work investigating the compared expressiveness of the comprehension and optic languages, and, particularly, the relationship between their major combinators (e.g. *flatMap* vs. vertical composition). Last, we would also like to investigate the role of optic algebras [19] in connection with the formalisation of the intended interpreters for optic representations.

On the implementation side, we found the typed final approach specially suitable to encode the separation of concerns between declarative optic representations and their intended queries. Optic combinators make up the bulk of the optic DSL, whereas intended queries provide their *observations* or interpreters. Other essential feature from the tagless final pattern we plan to profit from is extensibility. In particular, new optics will be added to the language through their own type classes (as we did for getters, affine folds and folds) so that we can fully reuse old queries without recompiling sources.

Optics as a modeling language for LINQ. By lifting optics into a full-fledged DSL, we opened the door to non-standard representations that translate the language of optics to data accessors for alternative formats beyond immutable data structures, such as XML documents or data stored in relational databases. In other words, we enabled an effective use of optics within the realm of language-integrated query, which was the main motivation of this work. We argue that optics has a number of benefits over comprehensions, the current language of choice for LINQ, as a modeling language. As we showed, in adopting the language of optics, the programmer remains unaware of variables and keys, which results in more simple queries. Particularly, this is the query that remains to complete Table 1.

```
def under50_d[Repr[_] : Symantics : CoupleModel]: Repr[Fold[CoupleN, String]] =
  couples ▹ her ▹ filtered (age < 50) ▹ name
```

The optic version is simpler in two major respects: firstly, it avoids the declaration of variables; secondly, there is no need at all for equality checks over keys.

This last feature is a consequence of the nested character of optic models. The advantages of nested data models are also acknowledged in [7], where they are used to improve the readability of queries. In fact, our optic-based version of the *expertise* query in that paper is no more simple as their nested version. However, in order to get there flat representations have to be manually transformed into nested ones, and standard combinators *all*, *any*, etc., have to be redefined in terms of comprehensions. On the contrary, by embracing the language of optics, we work with nested models by default and have such combinators readily available. In fact, optics libraries in the folklore, such as Haskell *lens*[16], offer a huge catalogue of combinators that we may use to extend Stateless.

Moreover, the language of optics readily offers a rich type system to represent the cardinality results of queries. For instance, we may express a *getAge* optic that points at the age of a person whose name is taken as a parameter, and therefore might not be present in the state.

```
def getAge[Repr[_] : Symantics : CoupleModel](s: String): Repr[AffineFold[Person, Int]] =
  filtered(name === like(s)) ▹ age
```

As you can see, the previous definition returns a representation of an affine fold, where it is clear that the number of pointed values is at most one. In the comprehension approach, that query would normally return a list of values, since list is commonly the only standard semantic domain of comprehension-based LINQ languages. Moreover, note that `getAge` returns a *parameterized* representation. Following recent trends in the typed final approach (e.g. the language `SQUR` [18],

[16] http://hackage.haskell.org/package/lens

[17]), Stateless is first-order, i.e. doesn't extend the lambda calculus[17]. As this example shows, this does not, however, prevent us from being able to model parameterized or dynamic queries, as the list of LINQ requirements in [7] mandates.

Concerning the limitations of the approach, the most fundamental one is that the relational model is more general than the nested one and therefore not every model is expressible in Stateless. Take the couple model as an example. Here, we assume that each person is hanging from a couple and therefore we can find them by diving into the couple fields `her/him`. However, the relational model is able to supply more entries for people who do not necessarily conform a couple. To alleviate this problem, we could have provided a virtual *root* type and have `couples` and a new fold `people` using it as source. Still, the connections between `people` and `her/him` would be unclear in the optic model, so new mechanisms should be introduced in order to establish the precise relationship among them (e.g. a constraint language to express that "all members pointed by `her/him` must be pointed by `people` as well").

Optics as a general query language. We have provided an interpretation to turn Stateless queries into XQuery expressions, where we have seen that the connection among them is straightforward, leading to a compositional interpreter. The translation ignores the XQuery FLWOR syntax and basically focuses on XPath features. Indeed, we understand XPath as a language to point at parts of an XML document, so it is ideal as a representation of optics. Particularly, since XPath does not provide the means to update an XML document, it fits perfectly with read-only optics such as getters, affine folds and folds.

It might be worth mentioning that synergies among optics and XML are not new at all. In fact, prominent optic libraries are extended with modules to cope with XML[18] or JSON documents, even packaged as domain-specific query languages, such as JsonPath [19]. In these projects, standard optics facilitate the definition of these DSLs for querying Json or XML. Our approach is radically different, since we provide a general optic language to build generic optics which may be translated over those DSLs (JsonPath, XQuery, etc.). Our approach also differs from others where the process is reversed and a translation of XPath expressions into a general query language based on comprehensions is performed [7].

We have also provided a SQL interpreter, which is the primary target of classical LINQ with comprehensions. Commonly, comprehension-based queries need to be flattened to guarantee good performance: the naive translation to SQL is not optimal at all, since they typically lead to nested subqueries. Moreover, translations to SQL seek to be total and avoid the problem of query avalanche. Our translation to SQL attains similar guarantees, resulting in single queries without subqueries, beyond the ones generated by **`exists`** – which are unavoidable. The normalization process of comprehension queries is performed either through syntactic re-writings, as in T-LINQ [7, 8], or semantically as in SQUR [18].

Our translation process resembles this latter semantic approach, in that we use an intermediate language *TripletFun* to decouple the filtering, selection and collection aspects that will make into the final SQL query. We differ, however, in that the ultimate translation to SQL is performed directly from this non-standard semantics, rather than from a normalised optic query. We plan to incorporate normalisation and partial evaluation in future work, which will be convenient as soon as we extend the language with projections `first` and `second`, in correspondence with the `fork` combinator. Moreover, the purpose of our intermediate language is not to achieve a more optimal

[17] It would be straightforward to extend the language with first-order functions in case we need it, e.g. if we want to serialize/deserialize parameterized queries.

[18] https://hackage.haskell.org/package/xml-lens-0.1.6.3/docs/Text-XML-Lens.html

[19] https://github.com/julien-truffaut/jsonpath.pres

translation, but make it possible at all – a direct translation to SQL from the optic expression seems unattainable.

Connections between optics and databases are widespread. Actually, lenses emerged in this context [10] under the umbrella of *bidirectional programming*. We remark [14] as a recent work in this field, where a practical approach to the view update problem is introduced, by means of so called *incremental relational lenses*. Although we still do not know if extending Stateless will lead us to contemplate views in the non-standard SQL semantics, we find this research essential to deal with updating optics in an effective way.

7 CONCLUSIONS

This paper has attempted to show that optics embrace a much wider range of representations beyond concrete, van Laarhoven, profunctor optics and other isomorphic acquaintances. We showed, for instance, that a restricted subset of XQuery can be properly understood as a non-standard *optic*, i.e. as a representation whose essential purpose is allowing us to refer to parts of a data source using powerful combinators, declaratively, and derive effectful queries from those pointers. From this standpoint, data sources of optic representations may range far beyond general immutable structures: they might be XML documents, as in the case of XQuery, or relational databases. In fact, we also showed how to derive SQL queries from an intermediate optic representation specially designed to make that translation feasible. Strictly speaking, SQL is not an optic but a query language which is translatable from an optic representation. This representation, that we named *TripletFun*, might be understood as a "relational" optic representation, since it aims at pointing at parts of an abstract relational data source. In future work, we aim at challenging the generality of the language of optics through the generation of other optimal, idiomatic translations into a diverse range of querying infrastructures. We will pay attention in particular to more recent technologies than XQuery with a clear bias towards nested data models, such as document-oriented NoSQL databases and languages like GraphQL [13].

Technically speaking, we formalized the language of optics in terms of a tagless final encoding of a full-fledged DSL, that we named *Stateless*. Concrete optics provide the denotational semantics of this language, and its type system encodes the compositional structure of optics at a general level of abstraction, without being committed to particular representations. Stateless is thus formalized as a type class: the class of optic representations. Concrete optics serve as the standard representation, whereas XQuery or TripletFun can be understood as non-standard optics. Currently, Stateless only pays attention to a very restricted set of read-only optics, namely getters, affine folds and folds. In future work we will pay attention to other writable optics, like lenses, affine traversals or traversals, and additional combinators that populate de-facto libraries like Haskell lens and Monocle. This will force us to pay attention as well to the formalisation of the non-compositional features of optics: the different queries that they are intended to support and their accompanying laws (e.g. the get-set law of lenses). We think optic algebras [19] will be instrumental in that formalisation.

The ultimate goal behind this quest for the language of optics was showing that the theory and practice of language-integrated query can also build upon optics, besides list-comprehensions. Both approaches deserve their own merits. On the one hand, the native nested data models endorsed by optics lead to more declarative and readable queries, and foster the reuse of standard combinators; moreover, optics show a potential to cope not only with retrieval queries but also with updates, a kind of queries which are commonly neglected in theoretical accounts, but patently necessary in practice. On the other hand, the flat structure endorsed by comprehension-based languages like NRC, T-LINQ or SQUR, are more apt to cope with complex, non-hierarchical models; moreover, there are some combinators with a monadic component that are easily expressible with comprehensions but turn out to be weird in the optic world. We leave to future work elucidating the specific subset

of comprehension queries that can be derived from the declarative combinators of optics, with particular attention to recent work on comprehensions based on graded monads [12]. Similarly, we plan to extend this set of optic combinators with new components, such as horizontal composition for folds, which is fundamental to generate queries that produce cartesian products as result.

Finally, the current implementation of Stateless in the Scala programming language was developed as a proof-of-concept. Similar implementations may have been developed in other languages featuring higher-kinded types such as Haskell or OCaml. The results obtained are encouraging enough to anticipate the feasibility of a new generation of optic libraries that rival existing comprehension-based libraries for LINQ.

A XML SCHEMAS

A.1 Couple XSD

```
<?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>
```

A.2 Organization XSD

```
<?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>
```

ACKNOWLEDGMENTS

This work is partially supported by a Doctorate Industry Program grant to Habla Computing SL, from the Spanish Ministry of Economy, Industry and Competitiveness.

REFERENCES

[1] Flavio W. Brasil. [n. d.]. Quill - Compile-time Language Integrated Queries for Scala. ([n. d.]). https://getquill.io

[2] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Record* 23, 1 (1994), 87–96. https://doi.org/10.1145/181550.181564

[3] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (1995), 3–48. https://doi.org/10.1016/0304-3975(95)00024-Q

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[5] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543. https://doi.org/10.1017/S0956796809007205

[6] James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types.* Technical Report. Cornell University.

[7] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 403–416. https://doi.org/10.1145/2500365.2500586

[8] Ezra Cooper. 2009. 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 (Lecture Notes in Computer Science)*, Philippa Gardner and Floris Geerts (Eds.), Vol. 5708. Springer, 36–51. https://doi.org/10.1007/978-3-642-03793-1_3

[9] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. 2015. A clear picture of lens laws. In *International Conference on Mathematics of Program Construction*. Springer, 215–223.

[10] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. 2005. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices* 40, 1 (2005), 233–246.

[11] Apache Software Foundation. [n. d.]. The Cassandra Query Language. http://cassandra.apache.org/doc/4.0/cql/

[12] Jeremy Gibbons, Fritz Henglein, Ralf Hinze, and Nicolas Wu. 2018. Relational algebra by way of adjunctions. *PACMPL* 2, ICFP (2018), 86:1–86:28. https://doi.org/10.1145/3236781

[13] Olaf Hartig and Jorge Pérez. 2017. 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.*, Vol. 1912. Juan Reutter, Divesh Srivastava.

[14] Rudi Horn, Roly Perera, and James Cheney. 2018. Incremental Relational Lenses. *Proc. ACM Program. Lang.* 2, ICFP, Article 74 (July 2018), 30 pages. https://doi.org/10.1145/3236769

[15] Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4es, Article 196 (Dec. 1996). https://doi.org/10.1145/242224.242477

[16] Oleg Kiselyov. 2010. Typed Tagless Final Interpreters. In *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures (Lecture Notes in Computer Science)*, Jeremy Gibbons (Ed.), Vol. 7470. Springer, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

[17] Oleg Kiselyov. 2017. *Effects without monads: non-determinism.* Technical Report.

[18] Oleg Kiselyov and Tatsuya Katsushima. 2017. 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 (Lecture Notes in Computer Science)*, Bor-Yuh Evan Chang (Ed.), Vol. 10695. Springer, 364–383. https://doi.org/10.1007/978-3-319-71237-6_18

[19] Jesús López-González and Juan M. Serrano. 2018. Towards Optic-based Algebraic Theories: The Case of Lenses. In *Trends in Functional Programming*. Springer.

[20] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. 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, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 706. https://doi.org/10.1145/1142473.1142552

[21] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 25–36. https://doi.org/10.1145/2847538.2847541

[22] Russell O'Connor. 2011. Functor is to Lens as Applicative is to Biplate: Introducing Multiplate. In *7th ACM SIGPLAN Workshop on Generic Programming*. ACM.

[23] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. *ACM Sigplan Notices* 45, 10 (2010), 341–360.

[24] Matthew T Pickering, Nicolas Wu, and Jeremy Gibbons. 2017. Profunctor Optics: Modular Data Accessors. *Art, Science, and Engineering of Programming* 1, 2 (4 2017). https://doi.org/10.22152/programming-journal.org/2017/1/7

[25] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, Safely-extensible and Efficient Language-integrated Query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2847538.2847542

[26] P. Trinder and P. Wadler. 1989. Improving list comprehension database queries. In *Fourth IEEE Region 10 International Conference TENCON*. 186–192. https://doi.org/10.1109/TENCON.1989.176921

[27] Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 61–78.

[28] Philip Wadler. 1995. Monads for functional programming. In *International School on Advanced Functional Programming*. Springer, 24–52.

[29] Philip Wadler. 2002. XQuery: A typed functional language for querying XML. In *International School on Advanced Functional Programming*. Springer, 188–212.