DOI: 10.15439/2015F156
ACSIS, Vol. 5

A step towards genuine declarative language-integrated queries

Radosław Adamus
Institute of Applied Computer Science, Lodz University of Technology
90-924 Lodz, ul. Stefanowskiego 18/22, Poland
Email: r.adamus@iis.p.lodz.pl

Tomasz Marek Kowalski
Institute of Applied Computer Science, Lodz University of Technology
90-924 Lodz, ul. Stefanowskiego 18/22, Poland
Email: t.kowalski@iis.p.lodz.pl

Jacek Wiślicki
Institute of Applied Computer Science, Lodz University of Technology
90-924 Lodz, ul. Stefanowskiego 18/22, Poland
Email: j.wislicki@iis.p.lodz.pl

***Abstract*—Native functional-style querying extensions for programming languages (e.g., LINQ or Java 8 streams) are widely considered as declarative. However, their very limited degree of optimisation when dealing with local collection processing contradicts this statement. We show that developers constructing complex LINQ queries or combining queries expose themselves to the risk of severe performance deterioration. For an inexperienced programmer, a way of getting an appropriate query form can be too complicated. Also, a manual query transformation is justified by the need of improving performance, but achieved at the expense of reflecting an actual business goal. As a result, benefits from a declarative form and an increased level of abstraction are lost.**

In this paper, we claim that moving of selected methods for automated optimisation elaborated for declarative query languages to the level of imperative programming languages is possible and desired. We propose an optimisation method for collection-processing constructs based on higher-order functions through factoring out of free expressions in order to avoid unnecessary multiple calculations. We have implemented and verified this idea as a simple proof-of-concept LINQ optimiser library.

I. Introduction

SINCE the release of LINQ (Language-Integrated Query) for the Microsoft .NET platform in 2007, there has been a significant progress in the topic of extending programming languages with native querying capabilities [1]. Programming languages are mostly imperative; their semantics relies on the program stack concept. They operate on volatile data and the meaning of collections is rather secondary. On the other hand, query languages are usually declarative and their semantics often bases on some forms of algebras or logics; these languages operate mostly on collections of persistent data. Declarativity of a query language reveals itself mostly when considering operators for collections. In the case of an imperative language, operating on a collection takes a form of an explicit loop iterating over collection elements in a specified order, while in query languages one declares a desired result (e.g., a sub-collection containing elements of a base collection matching a given selection predicate) and an algorithm of filtration itself is not an element of an expression representing the query. Based on characteristics of data structures, a database state and existence of additional auxiliary structures (e.g., indices), an execution environment can choose the most promising algorithm (a plan) for evaluation of the query. Declarativity allows one to postpone selection of an algorithm even to the moment of an actual query execution. In this paper we discuss to what extent solutions for processing of collections within programming languages are actually declarative. To do so, we made an extensive research on query optimisation. In databases it is a crucial process that allows a programmer to be relieved from thinking about details of a processing control flow, auxiliary data structures and algorithms.

LINQ seems to be the most robust solution introducing a promise of declarative collection processing within an imperative programming environment. It is commonly used for direct processing of collections and as a mapper to resources devoid of a robust declarative query API or query optimisation. When encountering performance issues, developers are forced to manually optimise LINQ expressions or partly resign from declarative constructs in favour of an imperative code.

```
var ikuraQuery =
  from p in products
  where (
    from p2 in products
    where p2.productName == "Ikura"
    select p2.unitPrice).Contains(p.unitPrice)
  select p.productName;
```

Listing 1. Example 1 – query expression syntax.

Consider a LINQ query expression in Listing 1 (the database diagram including the Products table is available at *http://northwinddatabase.codeplex.com/*) whose purpose is to find names of products with a unit price equal to a price of a product (or products) named Ikura. If the query addresses a native collection of objects, its execution is severely inefficient as the nested subquery, searching for prices of products named Ikura, is unnecessarily evaluated for each product addressed by the outer query. Although this task could be resolved in a time linearly proportional to the collection's cardinality, the LINQ engine induces an outer loop and a nested loop, both iterating over the products' collection. Using this example, in further sections we show that manual optimisation of complex LINQ queries is not an easy task.

LINQ enables to express the same goal in many different ways. However, evaluation times of two semantically equivalent queries may differ by several orders of magnitude. In particular, in the context of the LINQ query expressions' declarative syntax, it violates the declarative programming principle. Without knowledge on how a query engine works in a context of given data, the optimisation process is too complex and time-consuming. This is particularly true if a programmer wants to preserve semantics and properties of his query construct.

To the best of our knowledge, the problem of automated global optimisation of LINQ queries for direct processing of collections of objects has not been addressed in the literature so far. By global optimisation we understand the ability to define an efficient query execution plan based on the whole query structure as opposed to the local optimisation that usually only targets a single operator. Below we prove that global optimisation can be done automatically making LINQ genuinely declarative.

Nonetheless, the problem that this paper deals with is not limited to LINQ. Surprisingly, it extends to dozens of programming environments that support functional-style operations on collections of elements, such as filter, map or reduce. Pipelines and streams introduced in Java 8 are a solution equivalent to LINQ to Objects [2]. The main difference lies in the naming convention of new operators corresponding to their functional prototypes (e.g., *map* and *filter* instead of LINQ's *Select* and *Where*). Furthermore, list comprehension constructs are examples of a shorthand syntax for specifying operations of projection and selection (filtering). Consequently, discussed issues concern many imperative languages exploiting this feature (e.g., Python). Fowler summarises such a functional-style programming pattern using a term collection pipeline [3]. Examples given in LINQ can be expressed in many imperative and functional programming languages. While we extend the conclusions of our work to the universe of imperative programming, they do not directly apply to functional languages (e.g., Haskell) since their principles of program evaluation are significantly different [4].

The rest of the paper is organised in the following way. First, we present a brief description of the state of the art followed by characteristics of language-integrated query constructs. Next, we describe issues with nested independent subqueries and free expressions revealing a huge optimisation potential. Finally, we present our solution followed by measured results and principles of our optimisation approach, being the core of the paper. The paper is concluded with a short summary.

II. Related Work and the State-of-the-Art

Databases are the area of the computer science where declarative programming and query optimisation have developed extensively. Over 40 years of the research on relational systems resulted in various optimisation techniques [5][6] and numerous solutions are incorporated in available commercial products. Our research presented in the paper particularly addresses query optimisations analogous to query unnesting, dating back to the early 80s [7]. This topic is constantly appearing in the context of arising database technologies. Different approaches to handle nested queries evaluation have been proposed for object-oriented databases [8][9] and XML document-based stores [10]. However, NoSQL solutions marginalise the topic of query languages and usually rely on a minimalistic programming interface and domain-specific optimisations, mostly implemented by high redundancies and storing data in the form matching assumed queries. Most attention from the scientific community concentrates on the topic of distributed data-parallel computing using the Map/Reduce paradigm (like Hadoop or Dryad for Azure). This paradigm can be transparently used in declarative collection processing. The Dryad programming environment based on LINQ [11] takes advantage of mechanisms similar to Map/Reduce in order to write scalable, parallel and distributed programs. To increase sharing of computations in a data centre, Dryad can benefit from the Nectar system [12]. It is able to cache results of frequently used queries and incrementally update them. The use of cached results is achieved through automatic query rewriting. Robust query and program optimisations have been developed for solutions based on the functional paradigm. According to Fegaras [13], an optimisation framework for a functional lambda-DB object-oriented database relies on mathematical bases, i.e. the monoid comprehensions calculus. It generalises many unnesting techniques proposed in the literature.

Glasgow Haskell Compiler (GHC) for the Haskell non-strict purely functional language introduces many methods based on code rewriting. They range from relatively simple rules that can be used to improve efficiency of programs through modifications on a high syntactic level to more complex low-level core language transformations (e.g., let-floating, beta reduction, case swapping, case elimination) [14]. In particular, a procedure called full laziness (or fully lazy lambda lifting) has been proposed to avoid reevaluation of inner expressions for which result could be pre-calculated only once [15][16].

Currently, due to introduction of lambda abstractions into object-oriented languages, functional style of programming became ubiquitous. Stream and collection processing constructs derived from functional languages can be naturally evaluated in parallel using multiple processor cores. Therefore, the most popular solutions, like Java 8 streams, LINQ or ScalaBlitz, enable such optimisation through various libraries or frameworks [17].

In the field of functional-style queries integrated into a programming language, the topic of query optimisation seems the most advanced in LINQ. A LINQ provider library can implement direct processing of data (e.g., LINQ to Objects, LINQ to XML) or delegate processing to a remote external resource by sending a translated query (e.g., LINQ to SQL, LINQ to Entities). To be precise, a mixture of both approaches can be used, e.g. when the query language of a remote resource cannot completely express the semantics of a LINQ query. In the case when LINQ sends a translated query, it also delegates the responsibility for query optimisation. Consequently, if the external resource engine provides optimisation, developers can fully rely on a declarative style of programming. However, in the context of LINQ to SQL, the problem of analysing and normalising of LINQ queries in order to provide minimal and cohesive mapping to SQL has drawn attention of the scientific community. This is caused mostly by some drawbacks of the original Microsoft's solution that in some cases may fail or produce a so-called "query avalanche" [18][19].

The issue of performance deficiencies while processing collections of objects has not passed unnoticed by the LINQ community. In order to cope with the shortage in optimisation comparing to database engines, the i4o project (abbr. index for objects) solution adapted the idea of indexing to native objects' collections [20]. It is implemented as an alternative for the LINQ to Objects provider library. Utilising the concept of secondary access structures, i4o can produce several orders of magnitude of a gain in performance for queries filtering data at the cost of a data modification overhead.

Another examples of LINQ query optimisation tools are Steno [21] and LinqOptimizer [22] provider libraries. Their authors focused on a significant performance deficiency of LINQ queries in contrast to the equivalent manually optimised code that can be several times faster. Experiments have shown that Steno allows one to obtain up to 14-fold increase in processing of sequential data and 2-fold comparing to a problem processed by the DryadLINQ distributed engine [11]. The main idea behind Steno is to eliminate the overhead introduced by virtual calls to iterators that are the fundamental mechanism used by the LINQ engine. This problem has been solved by automatic generation of an imperative code omitting iterators. The optimisation addresses mainly implementation of individual operators. This also concerns the case of nested loops' optimisation when Steno has to analyse a series of operators only to preserve the order of iteration induced by the LINQ to Objects library implementation. This is justified by the loop fusion efficiency and consideration of side effects that are allowed in LINQ. Steno is also capable of higher-level optimisation giving an example of the GroupBy-Aggregate optimisation. It involves a local term rewriting, addressing a pair of neighbouring operators, i.e. GroupBy followed by Aggregate. When encountering such a sequence of operators, Steno replaces it by a dedicated GroupByAggregate operator that saves memory by storing per-key partial aggregates instead of the whole collection of group values. This optimisation takes advantage of LINQ declarativity by changing the course of evaluation. As a result, introducing side effects would cause its incorrectness. Being aware of a difficulty of automatic reasoning about side effects within queries, Steno's authors suggest developer-guided optimisation. Optimisation similar to GroupBy-Aggregate is considered in the SkyLINQ project [23] that develops an alternative operator called Top. This operator can be used to substitute a sequence of OrderBy and Take method calls (i.e. an operation to get top k elements). The significance of LINQ grew up with introducing LINQ to Events, an extension enabling declarative programming according to the reactive paradigm [24]. The solution derives from Functional Reactive Programming and is well suited for composing asynchronous and event-based programs [25]. Recently, this approach has attracted attention of commercial and scientific communities and, as a programming paradigm, faces efficiency issues indicating possible areas for optimisation [26][27].

Other current research on LINQ strives to allow seamless integration of heterogeneous data sources [28]. As a result, users can transparently process and modify data shared among contributing resources. Because of complex multilayer architecture, such an environment is not efficiency-oriented. LINQ is generally focused on local optimisation performed at a data source layer. In processing of heterogeneous and distributed data, it is unlikely that such optimisation is provided by each contributing resource. Therefore, it raises a need for global optimisation performed at the level of a LINQ query itself.

Declarative functional-style constructs in general-purpose object-oriented languages are not pure. As a result, decisions concerning optimisation have to be made by programmers. Transparent and aggressive compile-time optimisations can be achieved by introducing a query language extension into a programming language compiler [29].

One of numerous examples of extending compilers of existing languages with declarative constructs is SBQL4J [30]. It enables seamless integration of SBQL queries with language instructions and executing them in a context of Java collections. SBQL4J is based on the Stack-Based Architecture (SBA) approach instead of the functional approach and offers capabilities comparable to the LINQ technology [31][32]. What distinguishes it from other programming language-integrated queries is incorporation of several automatic optimisation methods developed for SBA. One of these methods, i.e. factoring out independent subqueries [8], enables SBQL4J to cope with optimisation of queries equivalent to examples discussed in this paper. It belongs to the group of optimisation methods that are based on query rewriting. Factoring out concerns a subquery (that in SBA represent any subexpression) that is processed many times in loops implied by so called non-algebraic operators despite that in subsequent loop cycles its result is the same. In SBQL4J rewriting is applied at a compile-time and a resulting performance improvement can be very significant, sometimes giving query response times shorter by several orders of magnitude.

III. Characteristics of Language-Integrated Query Constructs

Declarative style programming (especially in the context of databases) is often associated with the select-from-where syntactic sugar known from SQL that was adapted into LINQ. The query in Listing 1 is expressed using the LINQ query expression syntax. That form lacks explicit information on an order of performed operations and virtually a compiler could translate it to any semantically equivalent lower-level code that could be considered a query execution plan. Consequently, programmers must be particularly careful about potential side effects within declarative constructs in order to avoid the risk of unpredicted violations. Technically, query expressions are syntactic sugar over the implementation layer using lambda expressions, higher-order functions and, so called, extension methods [33]. An executable query, after removing the LINQ syntax sugar, will take the form presented in Listing 2.

```
var ikuraQuery = products.
  Where(p => products.
    Where(p2 => p2.productName == "Ikura").
    Select(p2=>p2.unitPrice).Contains(
      p.unitPrice)).
  Select(p => p.productName);
```

Listing 2. Example 1 – de-sugared.

The translated query uses the traditional, non-declarative object-oriented programming syntax. When processing collections or XML documents directly, the most crucial LINQ library extension methods (e.g., Select and Where) expose iterators that perform a specified operation on elements of a given collection. Lambda expressions are used to express details concerning such an operation, e.g. the selection predicate for the Where operator. Despite of similarity of Listing 2 to the original query expression, such composition of method calls on the products collection determines the order of evaluation.

Due to the specific implementation based on iterators and lambda abstractions, the execution strategy of LINQ queries is deferred. Execution is performed in presence of functions or instructions forcing iteration over elements specified by a query. However, a result of an iteration is not saved or cached, so each execution reevaluates a query against a given (current) data state.

The approach used in the LINQ to the objects' library implementation is generally ubiquitous (however, not uniform) in numerous programming languages (e.g., Python, Java 8, Elixir, Ruby) [3]. A good summary describing the possible set of properties of functional-style constructs can be found in the documentation of Java 8 streams [2].

The functional nature makes a language construct a good candidate for optimisation due to intelligible querying. All operations in a query processing chain produce a new queryable result instead of modifying original data; hence they do not introduce side effects. For example, during filtration on a list, no element is actually removed. Even though filter and map (common functional-style operators) are often used to directly process elements of a local in-memory collection, in reality elements can be obtained one by one from any so called queryable data source, e.g., a data structure, a generator function, an iterator, an I/O channel and a chained pipeline of collection operations. Such generality allows, usually time-consuming, querying of remote data sources, additionally making optimisation desirable.

The above properties are common in implementations of language-integrated query mechanisms. However a programmer must be sensitive to possible differences in various programming languages. For example, some languages implement consumable evaluation of queries. In such a strategy, elements of a queryable data source instance can be visited only once during its life. As a result, each query instance can be evaluated only once. The last property is present in Java 8 streams whereas LINQ operators are not consumable. The laziness-seeking property has the most profound impact on evaluation and semantics of language-integrated queries. It is connected with the lazy evaluation strategy assuming that a next element is returned for further processing only if necessary. Usually, a place of a lazy construct definition does not determine an actual moment of query execution (i.e. deferred execution strategy). Actual query execution occurs when its result is required, for example elements referred by a query are iterated or counted. Operations like selection, projection, and removal of duplicates are often implemented lazily. Consequently, to ensure coherence execution of eager constructs (e.g., grouping or ordering) is also deferred. Lazy evaluation usually results in better performance. It is cache-friendly since an element is processed by a chain of collection pipeline operations before proceeding to the next element. Moreover, in cases when the desired query result has been reached before visiting all elements it is not necessary to continue iterating (e.g., a query finding the first product with name "Ikura").

In the context of query optimisation, it is important to preserve properties of optimised constructs. In a general case, any change in this matter can affect semantics of application code. Switching an expression evaluation strategy from lazy to eager or forcing immediate execution can have serious consequences. Only laziness-seeking constructs can deal with possibly unbounded data sources. Eager evaluation of selection and projection operators on an infinite data source would require infinite computational resources and time, while lazy evaluation can return partial results.

In the next section, we show that preserving deferred execution, which is implied by the lazy evaluation strategy, is the factor impeding query optimisation.

IV. Performance Pitfalls

A. Evaluation of Independent Subqueries

Analysing the expression from Listing 2, it becomes obvious that the nested query selecting products named Ikura will be executed multiple times, since it is a part of a lambda abstraction (specifying a selection predicate) called against each product (the external *Where* operator induces a loop iterating over elements of products collection). This form is not efficient and makes the computational complexity quadratic (i.e. $O(n^2)$). However, searching for products named Ikura is independent of the parent query and could be evaluated just once. In order to improve query performance, a programmer must transform it. A natural way for optimisation seems to be factoring out the problematic subquery to a separate instruction and assigning it to a new variable (see Listing 3). The changes could also be presented on the LINQ query expression, but because the form with extension methods is actually executed, it will be a basis for this study.

```
var nestedQuery = products.
  Where(p2 => p2.productName == "Ikura").
  Select(p2=>p2.unitPrice);
var ikuraQuery = products.
  Where(p => nestedQuery.Contains(p.unitPrice)).
  Select(p => p.productName);
```

Listing 3. Example 1 – loops in separate instructions.

A result of the transformation shown in Listing 3 may seem effective; however, the expected goal will not be achieved. The problem lays in the execution strategy of LINQ queries. The *nestedQuery* variable holds an instance of a non-executed query that will be evaluated – like in the case of the non-transformed expression (Listing 2) – at every traversal of the loop induced by the *ikuraQuery Where* operator.

In Java 8 streams proper execution of corresponding queries generally would become impossible due to the consumable property of streams. After the transformation, the selection predicate of the *ikuraQuery* would share the same instance of a *nestedQuery* stream. Evaluation of the nested query would be performed only once, at the first traversal of the loop induced by the *ikuraQuery Where* operator, whereas the following iterations would result in terminating query evaluation and throwing an exception.

Solving the above problems requires eliminating deferred execution of the nested query. There exist several techniques to force immediate execution of a LINQ query. For example, the *ToList* method returns a list containing a materialised query result. Applying it to the nested query makes the solution more efficient (linear computational complexity) than the query in Listing 2. However, a part of the original query is executed and the other part remains deferred to the moment of an actual demand. It is possible that data in a collection may change between creation of a query and its evaluation. The original query form (and the programmer's intention) is insusceptible to it – the query is always completely executed on a current data state. After immediate execution of the nested query one cannot be sure about it – the *ikuraQuery* can be evaluated when data needed for calculating the *nestedQuery* subquery got already modified. As a result of the transformation, there occurred a change of the query semantics that is very difficult to detect by a programmer or tests. Ultimately, a programmer is forced to resign from deferred execution of the whole query, which is shown in Listing 4.

```
var nestedQuery  = products.
  Where(p2 => p2.productName == "Ikura").
  Select(p2=>p2.unitPrice).ToList();
var ikuraQuery = products.
  Where(p => nestedQuery.Contains(p.unitPrice)).
  Select(p => p.productName).ToList();
```

Listing 4. Example 1 – fully immediate execution.

Due to explicit materialisation, reusing the optimised query against a different data state becomes troublesome. For an inexperienced programmer, a way of getting an appropriate query form can be too complicated. Without deeper knowledge on the LINQ internal semantics in a context of object data, obtaining an optimal structure of code is a tricky, time-consuming and error-prone task. The example shows a lack of real independence of LINQ from a type of a data source. Despite the fact that LINQ allows unified processing on various types and sources of data, an actual execution plan relies on them. It seems that the basis for elaborating this layer of the language was mostly the integration of the object-relational mapping with a type system of a programming language (what also shows at the level of the LINQ query expression syntax and implementing providers [30]).

B. Factoring Out Constructs Executed Immediately

Although LINQ queries execution is deferred, an execution strategy of some expressions comprising LINQ queries can be immediate. Such expressions are evaluated locally in the place of the definition. Some operator, like aggregate functions returning a single value instead of a queryable data source, force immediate execution of a query. The query in Listing 5 contains such an expression determining the greatest unit price in the products' collection. However, it will not be evaluated until the execution of the *maxQuery* since it is contained in a lambda expression defining a selection predicate for the *Where* method.

```
var maxQuery = products.
  Where(p => products.
    Max(p2=>p2.unitPrice) == p.unitPrice).
  Select(p => p.productName);
```

Listing 5. Example 2 – extension methods syntax (quadratic computational complexity).

Similarly to the subquery determining the price of the Ikura product, it should be evaluated only once during execution of the *maxQuery* and therefore needs to be factored out. Let us call such constructs free expressions. Using the same procedure as presented in the previous section, we break the query into two instructions and perform immediate execution (see Listing 6).

The *ToList* operation does not need to be applied to the expression defining maxPrice (actually, it cannot be applied because it returns a value), due to its inherent immediate execution.

```
var maxPrice = products.Max(p2=>p2.unitPrice);
var maxQuery = products.
  Where(p => maxPrice == p.unitPrice).
  Select(p => p.productName).ToList();
```

Listing 6. Example 2 – immediate execution (linear computational complexity).

C. Consequences of Changing the Evaluation Order

There exist some subtle consequences concerning evaluation after the manual optimisation. In the original forms of example queries (Listing 2 and Listing 5), nested expressions would be evaluated only if the products' collection is not empty. In the optimised forms (Listing 4 and Listing 6) it will be unnecessarily evaluated also when the collection is empty. This is particularly important for performance when a nested query operates on a collection different than the external query does. The current example concerns just one collection, but it is easy to imagine a situation when collections are distinct (e.g., products from other shops kept in separate collections). In extreme cases, if calculation of a factored out expression is time-consuming, this can worsen overall query performance. Aside from performance issues, the transformation presented in previous sections can have dangerous impact on query semantics. In the second example (Listing 5) in the case of an empty collection of products, the selection predicate is not evaluated at all and the final result is simply an empty collection of product names. After optimisation (Listing 6), the expression *products.Max(p2 => p2.unitPrice)* is always evaluated at the beginning. The *Max* method applied to an empty collection throws an exception. Consequently, the behaviour of the optimised query is unsafe and inconsistent with the intent of a programmer.

To make optimisation immune to the described risk, the original order of evaluation should be restored. This could be achieved by applying the lazy loading pattern to the free expression determining *maxPrice*. In Listing 7 we introduce an improved transformation.

```
var maxPriceThunk =
  new Lazy<Double>(products.Max(p2=>p2.unitPrice));
var maxQuery = products.
  Where(p => maxPriceThunk.Value == p.unitPrice).
  Select(p => p.productName).ToList();
```

Listing 7. Example 2 – immediate execution
(linear computational complexity).

A *Lazy* class instance is a simple thunk – an object in memory representing an unevaluated (suspended) computation, used in the call-by-need evaluation strategy. The argument of the Lazy constructor specifies a function that should be evaluated at most once, only if its value is requested for the first time. The request is signalised by accessing the *Value* property of the *Lazy* instance. Consequently, the original order of the query and the free expression evaluation are restored (except the free expression being processed at most once) making the optimisation semantically safe. It is achieved at the expense of overhead concerning access to the *Value* property.

In the next sections we present a general approach to optimisation that preserves semantics and characteristics of an original query while reducing its computational complexity.

V. Factoring Out Free Expressions

The solutions presented for both examples share a common shortcoming; they do not preserve the deferred execution property. Our main aim is to propose a general query rewriting rule overcoming the problem. In order to keep the solution generic, additional constraints have been assumed: (1) transformation should not break a query into separate instructions (in contrary to what is shown, for example in Listing 7), (2) we express the rules in general terms rather than LINQ specific ones (e.g., operators common in functional programming). Obviously, we assume also that the intent of a programmer is simply to query and not to introduce side effects deliberately.

Generalisation of the factoring out procedure should take into account queries more complex than presented above. A nesting level of a lambda expression in the examples presented in Listing 2 and Listing 5 is shallow, but conceptually can be arbitrary with no need to modify the factoring out procedure. Free expressions can be bound either globally, i.e. to an environment independent from a query, or to a lambda expression at any nesting level lower than the lambda expression containing a free expression. The examples presented in Listing 2 and Listing 5 concern the former case. A generalising solution to the latter case can be achieved by treating any subquery as a separate query and the rest as a global environment.

The basic idea behind the transformation is, first, to identify free expressions that could be evaluated before a loop induced by an operator containing them, and next, to apply an appropriate rewriting rule. This is generalisation of the standard procedure called loop-invariant code motion known from the compilation theory [34]. An example of incorporating this idea to programming language-integrated queries can be found in the Stack-Based Architecture theory [8]. To optimise evaluation in functional languages, a similar procedure of fully lazy lambda lifting (called also full laziness) has been also proposed [16]. In both cases rewriting rules are straightforward and make use only of the basic set of language operators. Our attempt to generalise, in a similar manner, factoring out free expressions within LINQ queries using only methods supplied by LINQ has been unsuccessful. In particular, LINQ operators in presence of a queryable data source (e.g., a collection) cause iteration over elements, whereas factoring out requires treating empty, single or multiple elements as an individual result cached for reuse in further calculation.

A. Formalising Optimisation

The procedure of factoring out free expressions can be applied to the following query pattern:

queryUnoptimized ::= *queryExpr*(λ(***freeExpr***))

where *queryExpr* denotes a query expression that includes a nested lambda abstraction λ(*freeExpr*) containing *freeExpr* that is free from any lambda abstraction within the query. Additionally, we assume that *freeExpr* should be evaluated several times during the execution in order to make factoring out profitable. This pattern is not restricted to a whole query. It can match any subquery.

The solution requires introducing transformation of factoring out a free expression before a loop using it and applying the lazy loading evaluation strategy. Several aspects need to be addressed to make such optimisation effective and general. (1) In imperative programming languages deferring execution is often achieved through enclosing code in a function or by introducing an iterator. In both cases, repeated execution (e.g., inside a loop implied by map or filter collection pipeline operators) causes repeated evaluation. If this applies to a factored out expression, then it is usually necessary to force materialisation of its result before entering a loop using it. (2) Moreover, materialisation solves the issue with factoring out consumable data sources since they cannot be evaluated more than once. Before factoring out, the problem does not exist since such constructs reside inside lambda abstractions (that are parameters of collection pipeline operators) and therefore are evaluated only once during single lambda call evaluation. (3) As stated earlier, it is possible that a free expression is skipped during evaluation of the original query. In a general case it is safe to preserve an order of evaluation by suspending materialisation of a factored out expression and prevent its immediate execution before entering a loop using it. (4) An instance of a mechanism used for suspending materialisation of a factored out expression should not be shared between query executions. To solve this problem, it can be additionally enclosed in a lambda abstraction. Otherwise, following executions would share a cached result determined during the first execution. (5) In order to prevent collection pipeline operators from iterating over a collection, it has to be nested into a new collection as a single element. The same procedure can be applied to a single result to enable usage of collection pipeline operators.

Let us denote the following abstract operations: ***Collection***(*arg*) – creates and returns a collection consisting of one element specified by an argument, e.g., if an argument is a collection it returns a nested collection), ***Immediate***(*expr*) – evaluates and materialises a result of an expression passed as an argument (except when the *expr* execution strategy is already immediate), ***Suspend***(*lambda*) – returns an instance of a mechanism for lazy loading of an expression specified by a lambda abstraction passed as an argument, ***Value***(*lazy*) – returns a lazily initialised value stored by a lazy loading mechanism instance specified by an argument.

Taking advantage of above operations, we introduce the following rewriting procedure:

queryOptimized ::=
Collection(() => ***Suspend***(() => ***Immediate***(*freeExpr*))).
map(*lambdaParam* => *lambdaParam*()).
map(*freeExprThunk* => *queryExpr*(
λ(***Value***(*freeExprThunk*)))).flatten()

where λ(***Value***(*freeExprThunk*)) is a nested lambda abstraction λ(*freeExpr*) with an occurrence of *freeExpr* expression substituted by ***Value***(*freeExprThunk*). This form ensures that execution of all components of the original query is deferred assuming that collection pipeline operators map and flatten have such an execution strategy.

The first part of the rewritten query

Collection(() => ***Suspend***(() => ***Immediate***(*freeExpr*)))

creates a collection consisting of a single element that is a lambda function creating an instance of a mechanism for suspended materialisation of the factored out free expression. The following map operator ensures execution of the lambda function. As a result, the following map operator will process

queryExpr(λ(***Value***(*freeExprThunk*)))

expression only once for *freeExprThunk* assigned the lazily loaded cached value of *freeExpr*. Therefore, the result of evaluation of *queryExpr*(λ(***Value***(*freeExprThunk*))) is equal to the result of evaluation of *queryExpr*(λ(*freeExpr*)). The flatten operator eliminates an outer collection implied by the *Collection* operator. Consequently, the final result of the optimised query is taken from evaluation of the

queryExpr(λ(***Value***(*freeExprThunk*)))

expression.

B. Implementing Optimisation in C#

In C#, to simplify optimisation we introduce an auxiliary method *AsGroup* to take care of the *Collection* operation and suspended evaluation of a lambda expression returning a materialised value of the free expression. Listing 8 shows the implementation of the auxiliary operator.

```
static IEnumerable<TSource> AsGroup<TSource>(
                         Func<TSource> sourceFunc) {
  yield return new Lazy<TSource>(sourceFunc);
}
```

Listing 8. Auxiliary optimisation method.

The ***Suspend*** operation is achieved by a Lazy class constructor ***new*** *Lazy*<*TSource*>(*sourceFunc*). The ***yield*** return statement is a syntax sugar enabling creating a collection available through an iterator deferring any computations until iteration starts. In this way, a programmer avoids using a concrete type of a collection and enables a compiler to choose the best implementation on its own. *AsGroup* exposes an iterator that returns only one element, i.e. an instance of a mechanism for suspended materialisation of the factored out free expression. It is created directly before yielding replacing the projection *map*(*lambdaParam* => *lambdaParam*()). Consequently, the rewritten query in case of LINQ takes the following form:

LINQ-deferredQueryOptimized ::=
AsGroup(() => ***Immediate***(*freeExpr*)).
SelectMany(*freeExprThunk* =>
queryExpr(λ(*freeExprThunk.Value*)))

where the *SelectMany* LINQ operator substitutes map and flatten and *freeExprThunk.Value* realises the *Value*(*freeExprThunk*) operation.

The above transformation can be adapted to a situation when *queryExpr*(λ(*freeExprThunk.Value*)) is a construct executed immediately (e.g., when it returns a single value). In that case *SelectMany* needs to be replaced with two operations: Select realising projection and *First* responsible for flattening and immediate execution:

LINQ-immediateExpressionOptimized ::=
AsGroup(() => ***Immediate***(*freeExpr*)).
Select(*freeExprThunk* =>
queryExpr(λ(*freeExprThunk.Value*))).*First*()

The ***Immediate*** operation is required only in the case when *freeExpr* is a LINQ query deferred in execution. Explicit materialisation can be achieved using LINQ specific methods, e.g., *freeExpr.ToList*(). The transformation constitutes the general rewriting rule for optimisation of LINQ queries through factoring out free expressions. Applying it to the examples from Listing 2 and Listing 5 is shown in Listing 9 and Listing 10, respectively.

```
var ikuraQuery =
  AsGroup(() => products.
    Where(p2 => p2.productName == "Ikura").
    Select(p2=>p2.unitPrice).ToList()).
  SelectMany(ikuraPriceThunk => products.
    Where(p => ikuraPriceThunk.Value.
    Contains(p.unitPrice)).Select(p => p.productName));
```

Listing 9. Example 1 – after factoring out suspended free expressions optimisation.

```
var maxQuery =
  AsGroup(() => products.Max(p2=>p2.unitPrice)).
    SelectMany(maxPriceThunk =>
      products.Where(p =>
            maxPriceThunk.Value == p.unitPrice).
      Select(p => p.productName));
```

Listing 10. Example 2 – after factoring out suspended free expressions optimisation.

The queries execution strategy after optimisation remains deferred and in the case of the second example (Listing 10), the problem of the exception while addressing an empty products' collection does not occur.

VI. Performance Tests

We have evaluated the impact of factoring out of free expressions optimisation in C# by applying it manually to a number of problems: ***samePriceAs*** – given a collection of products, find products with the same price as the product specified by a name, ***maxPrice*** – given a collection of products, find products with the maximal price in the collection, ***promoProducts*** – given a collection of products, find names of products in the imaginary sale promotion, i.e. exactly *k* times more expensive than any other product, and ***pythagoreanTriples*** – from natural numbers between 1 and n find a number of triples satisfying the Pythagorean theorem.

In experimental tests, the collection of products ranged from 1 to 1,000,000 elements. The size of each product averaged to 175 bytes. Tests for ***samePriceAs***, ***maxPrice***, ***promoProducts*** and ***pythagoreanTriples*** problems have been conducted using queries in Listing 2, Listing 5, Listing 11, and Listing 12 accordingly. The problems have been solved relatively simply and each one has at least one free expression suitable for the factoring-out optimisation. Solutions to ***samePriceAs*** and ***maxPrice*** have free nested queries, whereas ***promoProducts*** and ***pythagoreanTriples*** introduce simple mathematical calculations that can be factored out. The tests include comparison with PLINQ and LinqOptimizer optimisation framework. We also combine them manually with our optimisation to explore limits and further opportunities.

```
var promoProducts =
  products.Where(p => products.
     Any(p2 => p2.unitPrice ==
             Math.Round(p.unitPrice / 1.2, 2))).
  Select(p => p.productName);
```

Listing 11. A query concerning the ***promoProducts*** problem before optimisation.

```
var pythagoreanTriples =
  Enumerable.Range(1, max + 1).SelectMany(a =>
    Enumerable.Range(a, max + 1 - a).SelectMany(b =>
      Enumerable.Range(b, max + 1 - b).Where(
        c => a * a + b * b == c * c))).Count()
```

Listing 12. A query concerning the ***pythagoreanTriples*** problem before optimisation.

We conducted our experiments on a workstation with a 4-core Intel Core i7 4790 3.6 GHz processor, 32 GB of DDR3 1600MHz RAM, hosting Windows Server 2012 R2. Benchmarks have been compiled for a x64 platform with enabled code optimisations using target .NET Framework v. 4.5. Tests results for following problems are presented in Fig. 1, Fig. 2, Fig. 3 and Fig. 4.

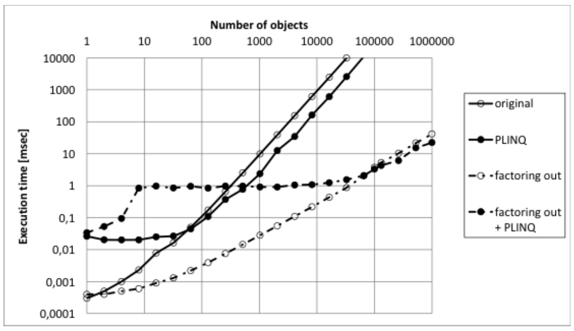

Fig 1. Query evaluation times for ***samePriceAs*** problem.

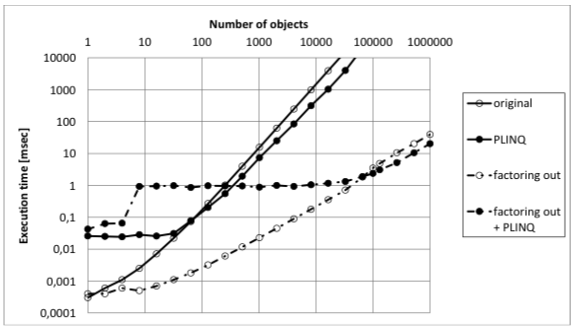

Fig 2. Query evaluation times for ***maxPrice*** problem.

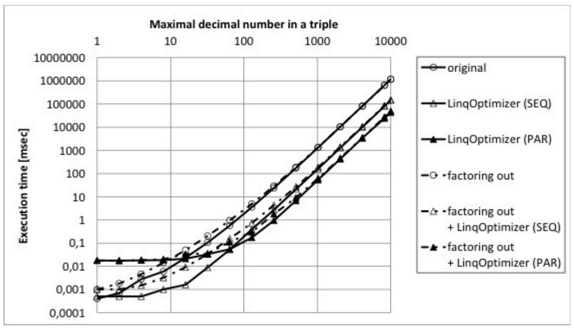

Fig 3. Query evaluation times for ***pythagoreanTriples*** problem.

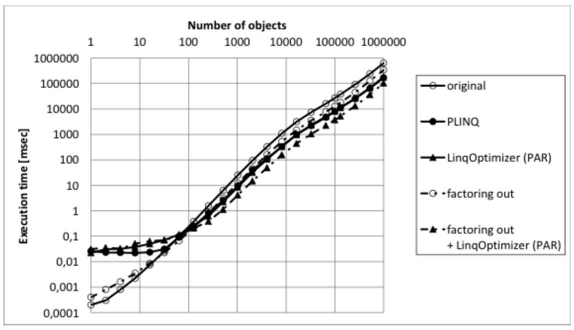

Fig 4. Query evaluation times for ***promoProducts*** problem.

The LinqOptimizer is used in two variants: sequential (denoted by SEQ) and parallel (denoted by PAR). The latter competes with PLINQ. Each query before and after factoring-out optimisation has been subjected to three further optimisation variants, i.e. PLINQ, LinqOptimizer sequential or parallel variant. The tests focus on query execution times and omit optimisation and compilation of a query. Most of the plots use logarithmic scales to more clearly reveal differences in performance for various collection sizes. To improve readability, the plots omit optimisation variants that are generally worse. In particular, the sequential variant of LinqOptimizer is shown only if it improved query performance in any collection size range, and the better alternative between PLINQ and parallel variant of LinqOptimizer is selected.

Results of the tests are as follows:

- Tests' results are consistent with an expected computational complexity. In ***samePriceAs*** and ***maxPrice*** problems it has been reduced from quadratic to linear, achieving a gain in orders of magnitude for large collections, e.g., in the case of the second example (Listing 5 and Listing 10) the query after factoring out is more than 30,000 times faster for 100,000 products (boost from ~115 s to ~3.8 ms).
- Except for the ***pythagoreanTriples*** problem, the profitability threshold of individual factoring-out optimisation is very low when comparing to PLINQ and LinqOptimizer. Even for a collection of 2 objects, optimised queries can work faster than original ones (e.g. ***samePriceAs*** and ***maxPrice***).
- The performance penalty in the case of a collection consisting of a single element is at most 0.6 μs which corresponds to a ~60% deterioration (the ***pythagoreanTriples*** problem).
- When processing large collections, the factoring-out transformation can give several times better performance by taking advantage of PLINQ (especially in the case of the ***promoProducts*** problem). For smaller collections, PLINQ imposes overhead significantly greater than factoring out.
- The ***pythagoreanTriples*** problem optimisation tests show that it may be difficult to obtain a significant gain when factoring out a simple expression (i.e. a * a + b * b). A ~3% gain is achieved for n equal to 10,000. • The LinqOptimizer framework seems to be designed for optimising queries involving numbers rather than complex objects. Only in the ***pythagoreanTriples*** problem optimisation, it outperforms both PLINQ and factoring out.
- In general, combining factoring out of free expressions with LinqOptimizer is not likely to produce the best solution. However, it seems that tuning of the LinqOptimizer algorithm should be possible. In the ***pythagoreanTriples*** problem, PLINQ is able to produce more efficient query after factoring out, whereas LinqOptimizer favours the original query. Unfortunately, the differences are too small to be seen on the plot.

C# libraries offer a *Lazy* class realising the Suspend operation, but considering performance, we have implemented our own lightweight version. We have experimented with different variants of performing *Collection*, *Suspend* and *Immediate* operations but the presented solutions generally resulted in performance better than others.

VII. Automatic Optimisation

A. Free Expression Detection

The transformation is justified by the need to increase effectiveness, which is achieved at the expense of reflecting the business goal. As a result, benefits from a declarative form and an increased level of abstraction are lost.

LINQ expression trees enable run-time analysing and dynamic building of LINQ queries [36]. This feature allows developing an optimisation method relying on rewriting of a LINQ abstract syntax tree. Automated detection of specified query patterns and transformation to an optimised form are required to make LINQ queries truly declarative. The previous part of this paper deals with the latter, i.e. the definition of efficacious rewriting rules for factoring out of a free expression. This section describes an algorithm for detection of free expressions within a query. The procedure does not address any details of implementation for the LINQ platform. It is general in terms of functional-style programming.

Let us establish a set of definitions concerning expressions and lambda abstractions (inspired by the definitions introduced by Hughes [16]):

- **Def. 1 (bound variables of lambda)**. An occurrence of a variable within lambda λ_A is bound to λ_A if and only if it is a parameter of λA_A,
- **Def. 2 (bound expressions of lambda)**. An expression within lambda λ_A is bound to λ_A if and only if it contains a variable bound to λA_A.
- **Def. 3 (native lambda of expression)**. The innermost lambda in which an expression *e* is bound is its native lambda. Let us denote this lambda $n\lambda(e)$,
- **Def. 4 (free expressions in lambda)**. An expression *e* within lambda λ_A is free in lambda λ_A if λ_A is nested in native lambda of expression *e*.
- **Def. 5 (maximal free expressions)**. A maximal free expression (MFE) is a free expression of some λ_A that is not a proper subexpression of another free expression of λ_A.

Additionally, to simplify definitions and the algorithm description, we assume that names of variables are unique. Moreover, we implicitly treat a whole query as a lambda abstraction with all free variables (constituting a global environment) as its parameters. In the case of examples from Listing 2 and Listing 5 native lambda of each MFE is the whole query.

From the definitions above, it follows that any MFE e free in a lambda λ_A can be determined before λ_A evaluation. Precisely, it could be determined anytime during evaluation of $n\lambda(e)$. The above statement is correct since:

1. *e* is a free expression (see definition 5).
2. λ_A is inside $n\lambda(e)$ (see definition 4).

3. *e* is not bound to λ_A (see definition 3).
4. *e* does not contain variables bound to λ_A (see definition 2).
5. λ_A call does not introduce any variable (parameter) required by *e* (see definition 1) that makes *e* independent from λ_A.

Consequently, it is possible to factor out the expression *e* from λ_A and evaluate it at the level of the $n\lambda(e)$ lambda.

The algorithm uses the standard depth-first search approach and detects all MFEs during a single pass through a query expression tree. Expression visitation focuses on finding its bindings that we define as a set of lambda abstractions declaring variables (usually as lambda parameters) used in the expression. This information is further used to determine bindings of its parent. Usage of lambda abstraction parameters determines whether an expression is free or bound. Therefore, it is necessary to handle information about names of the parameters and lambda abstractions to which they are bound. This is a task of an auxiliary map called binders. To correctly manage parameters' binding, the procedure specifically handles lambda abstractions and terminal name binding expressions.

While visiting lambda abstraction, the binders' map is filled with its parameters. They are visible only within the lambda abstraction. This sets the right context for the recursive visitation of the lambda body in order to detect free expressions bound specifically to the current lambda. Finally, the bindings set is returned to the lambda parent except for the current lambda that is removed (information on binding to the current lambda is not relevant outside).

The binders' map is used when visiting name-binding terminal expressions. These expressions consist only of an identifier name. If a name is found in the binders' map, a corresponding lambda is returned (as a single-element bindings set). If a name is not bound to any lambda, then it is assumed to be a globally free variable.

The described behaviour does not concern a name on the right hand side of a member access operator (e.g., field names). Such a name is bound locally to its left side, therefore field member access bindings are inherited from their left side expression. In general, bindings for remaining types of expressions are simply inherited from their children (a sum of the sets).

In the implementation nesting level annotations for lambda abstractions and variables are introduced to simplify the binding analysis. Expression bindings provide sufficient information to determine all MFEs and their native lambdas.

To exemplify the algorithm let us consider the promoProduct problem shown in Listing 11. The query in its optimised form is presented in Listing 13. The expression determining a price *Math.Round*(*p.unitPrice* / 1.2, 2)) is unnecessarily evaluated multiple times during execution of the inner loop implied by the Any operator. What distinguishes this and previous examples is that the transformation applies not to the whole query but only to the *Where* predicate. Additionally, the predicate is not a LINQ query but an expression returning a Boolean value. Therefore, *Select* and *First* methods were used instead of *SelectMany*.

```
var promoProductsOptimized =
  products.Where(p =>
    AsGroup(() => Math.Round(p.unitPrice / 1.2, 2)).
    Select(priceThunk =>
      products.Any(p2 => p2.unitPrice ==
                   priceThunk.Value)).First()).
  Select(p => p.productName);
```

Listing 13. Example 3 – rewriting inside lambda abstraction.

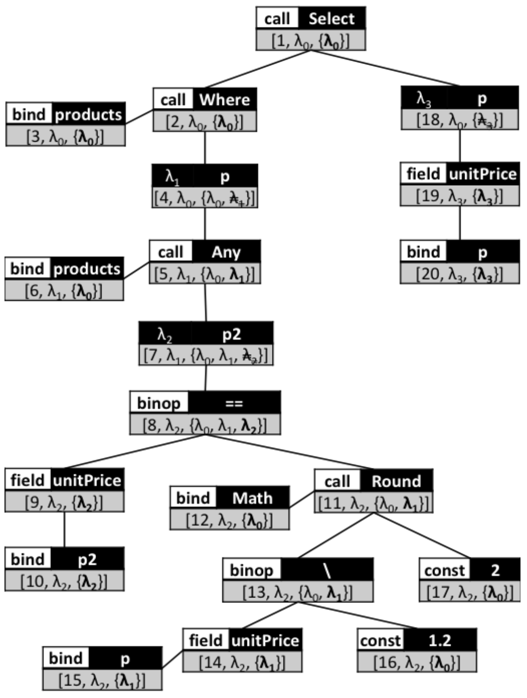

Fig 5. Example abstract syntax tree algorithm nodes annotations.

Partial results of the algorithm work for the unoptimised query are presented in Fig. 5. Each abstract syntax tree node of the query is annotated with three values: (1) a number indicating an order of visitation, (2) a lambda expression directly including an expression, (3) bindings set including the bolded element denoting a native lambda of an expression. Lambda expressions have been assigned unique numbers to facilitate their identification. Bindings that are removed at the end of lambda node visitation are indicated by a strikethrough symbol.

Free expressions have their native lambda (bolded lambda in bindings set) different from a nearest lambda (denoted by the second annotation), i.e. expressions with visitation order ranks 6, 11-17. After omitting terminal expressions such as literals (constant type nodes ranks 16 and 17) and name bindings (bind type nodes ranks 6 and 15), the only MFE left to factor out is *Math.Round*(*p.unitPrice* / 1.2, 2)). Its native lambda is λ_1. Hence, factoring out should be applied to its indirect parent: the *Any* node with a visitation order rank 5 (presented in Listing 11). It is an expression inducing iteration over the products collection at the highest level within λ_1.

B. Applying Factoring Out

The factoring out rewriting rule can be applied during visitation of lambda expressions. However, not all MFEs should be factored out. The conditions under which the optimisation promises well are described in analogous solutions [15][8], namely: (1) a free expression cannot be too simple (e.g., names and literals), (2) a free expressions' result should be used more than once. They can be verified during preparation to the transformation.

First, the complexity of an MFE can be examined. An appropriate threshold for applying transformation could be introduced, e.g., based on an arbitrarily set weight of language constructs comprising an MFE. Performance tests on the ***promoProducts*** problem involving factoring out a relatively simple expression have proven improvement in the case of collections consisting of at least 30 objects. For over 250 products optimised query was about twice as fast.

The second condition concerns a number of times that a MFE result is used in evaluation. An additional analysis may be necessary for confirming that $n\lambda$(MFE) contains a method that causes iteration over some collection that may require repeated evaluation of the MFE. For example, in LINQ this concerns mainly operators parameterised with a lambda abstraction (such as *Select*, *Where*, *Max*, etc). Operators operating on sets (e.g., *Contains*, *Union*) or custom ones are not any indication for the optimisation. The more detailed cardinality analysis is doubtful in case of a programming language environment and a lack of a cost model.

We have implemented a prototype LINQ provider library realising the mentioned optimisation (available at *https://github.com/radamus/OptimizableLINQ*). The analysis and the transformation are performed using the LINQ expression trees' representation available at runtime. Access to expression trees is provided though the *IQueryable*<T> interface that does not allow direct query execution. Instead, it exposes an abstract syntax tree of a query (in a form of a type-checked expression tree) to a data store provider. The provider makes use of this representation to build a query in a form (language) dedicated for a given data model (e.g., LINQ to SQL) [36].

Implementing optimisation in the form of a LINQ provider library gives a developer possibility to resign from aggressive, global query optimisation, e.g. when the order of evaluation is important considering some planned side effects. To enable automatic optimisation, the *AsOptimizable* extension method should be applied to a source collection. It is shown in Listing 14 for the Ikura product example.

```
var ikuraQuery = products.AsOptimizable().
  Where(p => products.Where(p2 =>
                    p2.productName == "Ikura").
    Select(p2=>p2.unitPrice).Contains(p.unitPrice)).
  Select(p => p.productName);
```

Listing 14. Example 1 – automatically optimised.

As a result, a rewritten query is compiled and becomes available for multiple use. One-time overhead occurring at the site of the definition is about a millisecond. A developer should consider runtime optimisation with caution when a query is used only once over a small collection. In contrast to LINQ, Java 8 streams operators are consumable, which prevents multiple usages of the same query. We are not aware of any mechanism enabling rewriting optimisations of Java 8 stream queries at runtime; nevertheless, in the case of consumable constructs the cost of optimisation done at runtime would burden each query execution.

VIII. Summary

The proposed solution proves that it is possible to provide programming languages offering functional-style access to querying data collections with resource-independent static optimisation mechanisms. We proposed a formal method – factoring out of free expressions – based on higher-order functions rewriting. Its essence is to avoid unnecessary recurring calculations. Factoring out of a free expression that is complex to calculate generally produces a robust performance gain. Such optimisation can be fully automated and does not require any interference or implementation-specific knowledge from a programmer. Using simple examples, we emphasise the significance of the order of evaluation implied by semantics of functional-style operators. Finally, we elaborate general and safe optimisation, considering characteristics of functional-style querying in imperative programming languages.

In contrast to the Nectar system [12], which also uses term rewriting to increase sharing of computations, our work addresses functional-style queries in general, i.e. without context of application which would limit our optimisation. We take advantage of the similar approach to optimisation as Steno [21], LinqOptimizer [22], or SkyLinq [23]. However, we make an attempt to explore more aggressive, global optimisations comparable to optimisations of database query languages.

The presented approach was verified in Microsoft .NET environment and its Language-Integrated Query technology. However, the automated solution has not been straightforward to elaborate due to necessity of considering several variants implied by execution strategies of constructs comprising LINQ queries and complexity of implementing LINQ providers.

Our optimisation for LINQ can be combined automatically with other ones as long as they preserve queries in an expression trees form. In other cases, fusion of optimisations has to be done manually. For example, PLINQ enables to take advantage of multiple cores and achieve several times better efficiency in processing of large collections. Moreover, the optimiser in some cases could automatically (or by a programmer's decision) resign from suspending evaluation of a factored out expression and remove overhead that it imposes. The tests showed that it results in further improvement of performance, up to ~18%. Finally, it seems that transformations would be the most profitable if incorporated in a compiler. Considering source-to-source transformations already performed by the C# compiler on LINQ query expressions [33] this solution imposes itself.

We believe that our work is as a real step towards genuine declarative language-integrated queries. We conduct further works on optimisation of functional-style constructs processing collections. One branch of our research concerns the elaboration of methods that are aware of operators semantics, e.g., addressing complex queries taking advantage of the selection operation, which exposes a huge potential for optimisation (e.g., pushing selection [37]). We also consider adapting other methods, such as revealing weak dependencies within queries that enable performing further factoring out [38].

References

[1] E. Meijer, “The world according to LINQ,” Commun. ACM, vol. 54, no. 10, pp. 45–51, Oct. 2011. http://dx.doi.org/10.1145/2001269.2001285

[2] Oracle, Java API docs, “Package java.util.stream”, http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, accessed: December 2014.

[3] M. Fowler, “Collection Pipeline”, 28 July 2014, http://martinfowler.com/articles/collection-pipeline/, accessed: December 2014.

[4] Hudak, "Conception, Evolution, and Application of Functional Programming Languages". ACM Computing Surveys 21 (3), pp. 383–385, 1989. http://dx.doi.org/10.1145/72551.72554

[5] Chaudhuri, “An Overview of Query Optimization in Relational Systems”, Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of data-base systems, Seattle, Washington, United States, pp. 34-43, 1998. http://dx.doi.org/10.1145/275487.275492

[6] M. Jarke, and J. Koch, “Query Optimization in Database Systems”, ACM Computing Surveys 16(2), pp. 111-152, 1984. http://dx.doi.org/10.1145/356924.356928

[7] W. Kim, “On optimizing an SQL-like nested query”, ACM Trans. on Database Systems, 7(3), pp. 443–469, 1982. http://dx.doi.org/10.1145/319732.319745

[8] J. Plodzien, and A. Kraken, “Object Query Optimization through Detecting Independent Subqueries”. Information Systems 25(8), pp. 467-490, 2000.

[9] S. Cluet, and G. Moerkotte, “Nested Queries in Object Bases”, In DBPL'93, pp. 226-242, 1993.

[10] N. May, S. Helmer, and G. Moerkotte, “Strategies for Query Unnesting in XML Databases”, ACM Transactions on Database Systems (TODS), Volume 31 Issue 3, September 2006, pp. 968-1013, 2006. http://dx.doi.org/10.1145/1166074.1166081

[11] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language”, Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, December 8-10, 2008.

[12] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. “Nectar: Automatic Management of Data and Computation in Datacenters”, In OSDI, 2010.

[13] L. Fegaras, and D.Maier, “Optimizing object queries using an effective calculus”, ACM Transactions on Database Systems (TODS), Volume 25 Issue 4, pp. 457-516, 2000. http://dx.doi.org/10.1145/1166074.1166081

[14] S. Jones, A. Tolmach, and T. Hoare, “Playing by the Rules: Rewriting as a practical optimisation technique in GHC”, Proceedings of the 2001 Haskell Workshop, pp. 203-233, 2001.

[15] S. P. Jones, W. Partain, and A. Santos, "Let-Floating: Moving Bindings to Give Faster Programs", Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, 1996. http://dx.doi.org/10.1145/232629.232630

[16] J. Hughes, “The Design and Implementation of Programming Languages”, Oxford University, D.Phil. Thesis, 1983.

[17] A. Biboudis, N. Palladinos, and Y. Smaragdakis, “Clash of the Lambdas”, 9th ICOOOLPS (Implementation, Compilation, Optimization of OO Languages, Programs and Systems) workshop, Uppsala, Sweden, 2014.

[18] T. Grust, J. Rittinger, and T.Schreiber, “Avalanche-safe LINQ compilation”, Proceedings of the VLDB Endowment, Volume 3 Issue 1-2, pp. 162–172, 2010. http://dx.doi.org/10.14778/1920841.1920866

[19] J. Chaney, S. Lindley, and P Wadler, “A practical theory of language-integrated query”, ICFP '13 18th ACM SIGPLAN international conference on Functional programming, ACM SIGPLAN Notices - ICFP'13, Volume 48 Issue 9, pp. 403-416, 2013. http://dx.doi.org/10.1145/2500365.2500586

[20] i4o, “i4o - Indexed LINQ”, http://i4o.codeplex.com, accessed: September 2014.

[21] D. G. Murray, M. Isard, and Y. Yu, "Steno: Automatic Optimization of Declarative Queries", PLDI’11 June 4–8, San Jose, California, USA, 2011. http://dx.doi.org/10.1145/1993498.1993513

[22] N. Palladinos, and K. Rontogiannis., “LinqOptimizer: an automatic query optimizer for LINQ to objects and PLINQ”, http://nessos.github.io/LinqOptimizer/, accessed: December 2014.

[23] Sky LINQ, “Sky LINQ”, https://skylinq.codeplex.com, accessed: December 2014.

[24] The Reactive Manifesto, “The Reactive Manifesto”, http://www.reactivemanifesto.org, 23 September 2013, accessed: December 2014.

[25] E. Meijer, “Your mouse is a database”, Commun. ACM, 55(5), pp. 66-73, 2012. http://dx.doi.org/10.1145/2160718.2160735

[26] I. Maier, and M. Odersky, “Higher-Order Reactive Programming with Incremental Lists”, ECOOP 2013 – Object-Oriented Programming, Lecture Notes in Computer Science Volume 7920, pp. 707-731, 2013. http://dx.doi.org/10.1007/978-3-642-39038-8_29

[27] G. Schueller, and A. Begrend, “Stream fusion using reactive programming, LINQ and magic updates”, 16th International Conference on Information Fusion (FUSION), pp. 1265-1272, 2013.

[28] Y. Wang, and X. Zhang, “The Research of Multi-source heterogeneous Data Integration Based on LINQ”, Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on Computer Science and Electronics Engineering, pp.147-150, 2012. http://dx.doi.org/10.1109/ICCSEE.2012.437

[29] C. Reichenbach, Y. Smaragdakis, and N. Immerman. PQL, “A purely-declarative java extension for parallel programming”. ECOOP ’12, LNCS 7313, pp. 53–78, 2012. http://dx.doi.org/10.1007/978-3-642-31057-7_4

[30] E. Wcislo, P. Habela, and K. Subieta, “Implementing a Query Language for Java Object Database”, ADBIS 2012, pp. 413-426, 2012. http://dx.doi.org/10.1007/978-3-642-33074-2_31

[31] R. Adamus., P. Habela, K. Kaczmarski., M. Lentner, K. Stencel, and K. Subieta, “Stack-Based Architecture and Stack-Based Query Language”, ICOODB Proceedings of the First International Conference on Object Databases. Germany, Berlin, pp. 77-95, 2008.

[32] K. Subieta, “Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL)”, http://www.sbql.pl, 2011, accessed: December 2014.

[33] G. M. Bierman, E. Meijer, and M. Torgersen, “Lost In Translation: Formalizing Proposed Extensions to C#”, Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, pp. 479-498, 2007. http://dx.doi.org/10.1145/1297105.1297063

[34] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, “Compilers - principles, techniques and tools”, Pearson Education, Inc., 2006.

[35] O. Eini, “The Pain of Implementing LINQ Providers”, ACM Queue - Mobile Devices in the Enterprise, Volume 9 Issue 7, 2011. http://dx.doi.org/10.1145/1978542.1978556

[36] T. Petricek, “Building LINQ queries at runtime in C#”, http://tomasp.net/blog/dynamic-linq-queries.aspx/, 2007, accessed: December 2014.

[37] M. Drozd, M. Bleja, K. Stencel, and K. Subieta, “Optimization of Object-Oriented Queries through Pushing Selections”, ADBIS 2012, pp. 57-68, 2012. http://dx.doi.org/10.1007/978-3-642-32741-4_6

[38] M. Bleja, K.Stencel, and K. Subieta, “Optimization of object-oriented queries addressing large and small collections” IMCSIT 2009, pp. 643-650, 2009. http://dx.doi.org/10.1109/IMCSIT.2009.5352770