Translation of Java-Embedded Database Queries with a Prototype Implementation for LINQ

submitted by
Kaichuan Wen
20729335

supervised by
Prof. Dr. Ralf Möller
Dr. Miguel Garcia

Hamburg University of Technology
Software Systems Institute (STS)

March 16th, 2009

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, March 16th, 2009
Kaichuan Wen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

Nowadays, enterprise data can be found in different kinds of sources: from an in-memory array, to XML documents, to relational databases. Each kind of data source has its own data access model and query mechanism. This diversity not only requires programmers to learn and compose various queries individually (as part of a single business use case), but also leads to problems such as weak typing and tight coupling.

Progress has been made in recent years, by introducing a thin middle layer between data sources and programming language, which on the one hand provides a unified query mechanism over different data sources, and on the other hand achieves Object/Relational Mapping (ORM) by integrating queries themselves into programming language with features like compile-time type checking, navigational cursors and so on.

On this direction, Microsoft has introduced LINQ (Language Integrated Query) in Visual Studio 2008 and .Net Framework version 3.5, bridging the gap between the world of objects and the world of data. Queries formulated using LINQ can run against various data sources such as in-memory data structures, XML documents and databases. While some of these use different implementations under the covers, all of them expose the same syntax and language constructs. As a groundbreaking innovation, LINQ is yet limited to those programming languages supported by .Net Framework 3.5. Therefore frameworks such as NLINQ[1] have come about, aiming at providing LINQ functionality in older versions of Visual Studio (C# and VB .Net) by providing a LINQ grammar parser and a "LINQ To Objects" execution environment.

With a similar motivation but different implementation techniques, this project work attempts to embed highly expressive, functional-style database queries into Java. Although the query language supported in the reported prototype is (a large subset of) LINQ, the underlying approach is in general applicable to other data query mechanisms as well (XQuery, JPQL, to name a few examples).

[1]`http://www.codeplex.com/nlinq`

1.2 Outline and scope

The main use case supported by the prototype is as follows: (a) the developer enters a LINQ query as a String in Java source code; (b) a compiler plug-in gets activated, participating in the compilation task; (c) this plug-in parses, checks, and generates a series of Java statements that build the abstract syntax tree (AST) for the input query; and (d) these statements are added to the Java compilation unit being processed, reporting any well-formedness errors that might have been detected over the API for communication between compiler plug-in and compiler. The next subsections explore in more detail each component involved in realizing this use case.

1. Grammar for LINQ

A parser for the LINQ syntax is necessary to transform LINQ queries into structural syntax trees. For this, the parser generator ANTLR (ANother Tool for Language Recognition) is used, which can generate lexer and parser for a DSL (Domain-Specific Language) into Java or C#, in addition to providing a grammar visualizer. Listing 1.1 and Figure 1.1 show as an example the grammar rule `unaryExpression` and its grammar diagram:

```
unaryExpression returns [CSTExpr value]
        :       unaSta1=statement { $value = $unaSta1.
            value; }
        |       '!' unaSta2=statement
                 {$value = factory.createUnaryExpr();
                  ((UnaryExpr)$value).setHasBang(true);
                  ((UnaryExpr)$value).setDotSepPrimExpr(
                      $unaSta2.value);
                 }
        |       '-' unaSta3=statement
                 {$value = factory.createUnaryExpr();
                  ((UnaryExpr)$value).setHasMinus(true);
                  ((UnaryExpr)$value).setDotSepPrimExpr(
                      $unaSta3.value);
                 }
        ;
```

Listing 1.1: Grammar rule of unary expression

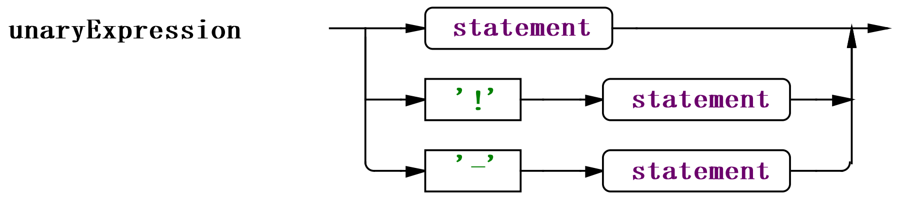
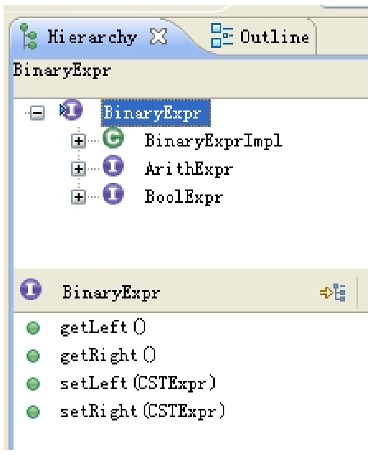


Figure 1.1: Grammar diagram of `unary expression`

Details about the LINQ grammar are described in Chapter 2.

2. CST metamodel

So that our query expander can represent each part and clause of LINQ syntax, a Concrete Syntax Tree (CST) metamodel is necessary. Modeling with EMF[2], class hierarchies as well as aggregations can reflect the relations and the interplay among different syntax elements[5]. For example, `BinaryExpr`, which represents a binary expression, should have two expressions as its left operand and right operand, as well as the setter and getter for them respectively. What's more, arithmetic expression and Boolean expression should be subinterfaces of binary expression. Figure 1.2 shows the type hierarchy of interface `BinaryExpr` (upper pane) and its members (lower pane):



```
BinaryExpr
  BinaryExpr
    BinaryExprImpl
    ArithExpr
    BoolExpr

BinaryExpr
  getLeft()
  getRight()
  setLeft(CSTExpr)
  setRight(CSTExpr)
```

Figure 1.2: Hierarchy and members of interface `BinaryExpr`

Once the metamodel has been finished, there should be a way to construct a CST (i.e., an object graph where nodes have been instantiated from the classes in the CST metamodel), according to what the parser matches from a query. ANTLR provides a mechanism of semantic action, which is ideal to accomplish this task. Semantic actions tell the generated parser what code should be executed when a specific production in the grammar is matched. The semantic action consists of executable statements of the target language (Java). In our case, semantic actions mainly consist of calls to factory methods as well as assignments of fields, resulting in CSTs where non-leaf nodes are instances of metamodel classes, and leaf-nodes are query tokens. Listing 1.2 shows the semantic action for grammar rule `groupClause`: (Semantic actions are those Java statements within a pair of curly brackets)

```
groupClause returns [GroupClause value]
        :       'group' grpSrc=expression 'by' grpBy=
            expression {
```

[2]http://www.eclipse.org/modeling/emf/?project=emf

```
                $value = factory.createGroupClause();
                $value.setSource($grpSrc.value);
                $value.setBy($grpBy.value);
        }
        ;
```

Listing 1.2: Grammar rule for `groupClause` and its semantic action

Details about CSTs and semantic actions used to build CSTs are given in Chapter 3.

3. Transforming CST into standard query operations

As specified in the C# Language Specification in Sec. 7.15[1], all textual LINQ query expressions are transformed into a series of method calls. For this, the C# Language Specification defines 18 transformation rules so that syntactically well-formed LINQ queries are guaranteed to be transformed into their corresponding series of method calls. Each of these transformation rules also defines its precondition (that a subtree of the CST must fulfill) in order for this rule to be applicable to it.

Details about why and how to transform concrete syntax tree into standard query operations, as well as some important aspects in the process of transforming are the focus of Chapter 4.

4. Conclusion

Chapter 5 gives a summary of the project and describes about what conclusions can be drawn, what has been learnt from the project, as well as the future work that can be done to extend the use case and functionality of this project work.

Chapter 2

LINQ Grammar

2.1 Original LINQ grammar

Query Expressions was introduced since .Net Framework 3.5, which provides a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery. Query Expressions is one of the two ways to compose a query (The other way is using *Standard Query Operators (SQO)[8]*, which will be introduced later).

Each query expression begins with a *from* clause and ends with either a *select* or *group* clause. The initial *from* clause can be followed by zero or more *from*, *let*, *where*, *join* or *orderby* clauses, which are called *query body* clauses.

A *from* clause defines the data source of a query or subquery and a range variable that defines each single element to query from that data source. The syntax of the *form* clause is:

```
from range-variable in data-source
```

Each *let* clause introduces a range variable representing a value computed by means of previous range variables, allowing to store the result of a subexpression that can be used somewhere else in the query. With *let* clause, the same expression that needs to be used many times in the same query does not have to be defined every single time. As expected, the syntax of the *let* clause is:

```
let name=expression
```

A *where* clause is a filter that excludes items from the result. It specifies a condition that an element in the data source must meet in order to be included in the results. A single query can have multiple *where* clauses or a *where* clause with multiple predicates that are combined by logical operators. Here is the syntax of the *where* clause:

```
where boolean-expression
```

Each *join* clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. The predicates *outerKeySelector* and *innerKeySelector* define how to extract the identifying keys from the outer and inner source sequence items. Here is the syntax of the *join* clause:

```
join range-var in inner
on outerKeySelector equals innerKeySelector
```

Each *orderby* clause reorders items by using one or more keys that combines different sorting directions. Directions can be either ascending or descending. Ascending is the default direction that applies when no direction is explicitly specified. Here is the syntax of the *orderby* clause:

```
orderby sort-on direction
```

The ending *select* or *group* clause specifies the shape of the result in terms of the range variables. A *select* clause specifies precisely what is obtained by the query, based on a projection that determines what to select from the result of the evaluation of all the clauses and expressions that precede it. A *group* clause projects a result grouped by a key, providing an effective way to retrieve data that is organized into sequences of related items. Following are the syntax of the *select* clause and *group* clause:

```
select expression
group range-variable by key
```

Finally, an *into* clause can, though this is not mandatory, be used to connect queries by treating the results of one query as a generator in a subsequent query. Here is the syntax of the into clause:

```
into name query-body
```

The other alternative to compose query besides Query Expressions is using *Standard Query Operators (SQO)*, defined as extension methods. In terms of .Net Framework, these methods can work with any object that implements either the *IEnumerable<T>* or *IQueryable<T>* interface.

In fact, in the .Net implementation of LINQ, all queries written in Query Expressions are translated by the language compilers into invocations of extension methods that are sequentially applied to the target of the query. For example, a LINQ query:

```
from c in customers where c.country == counties.Italy orderby c.name
descending, c.city select new {c.name, c.city}
```

is translated to the following method invocations:

```
customers.Where(c => c.country = Countries.Italy)
.OrderByDescending(c => c.name).ThenBy(c => c.City)
.Select(c=>new{c.name, c.city} .
```

The above method invocation contains two more new features: l*ambda expression* and *anonymous types.* Lambda expressions are more powerful syntax with which to write an anonymous method. Before the => token are parameters, which can be explicitly or implicitly typed. After the => is either a statement or an expression body. In the example above,

```
OrderByDescending(c => c.name)
```

is equivalent to

```
OrderByDescending(delegate (Customer c){return c.name}) .
```

Anonymous types has this general form:
```
new { nameA = valueA, nameB = valueB, ...}
```

The primary use of this syntax of object initialization is to create an object returned by the *select* clause. The outcome of a query is often a sequence of objects that is either a composite of two or more data sources, or a subset of the members of one data source. The type being returned is often needed nowhere else other than the query. In this case, anonymous type helps eliminate the need to declare a separate class just to hold the result of the query[6].

In our project, we support all these LINQ-related features: query expressions, query operators, lambda expressions and anonymous type.

2.2 Production rules for LINQ grammar

Representing LINQ grammar with ANTLR production rules is rather straightforward with ANTLRWork, a graphical development environment for developing and debugging ANTLR grammars. Figure 2.2 on page 8 shows an screenshot of ANTLRWork.

value

We first define the production rules for the most bottom elements of the syntax such as integer, float, string, identifier, and so on. These elements are then generalized into production of *value*, as shown in Figure 2.1

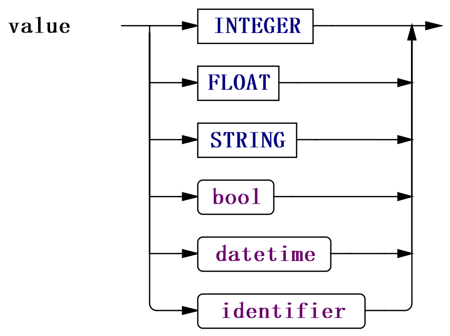
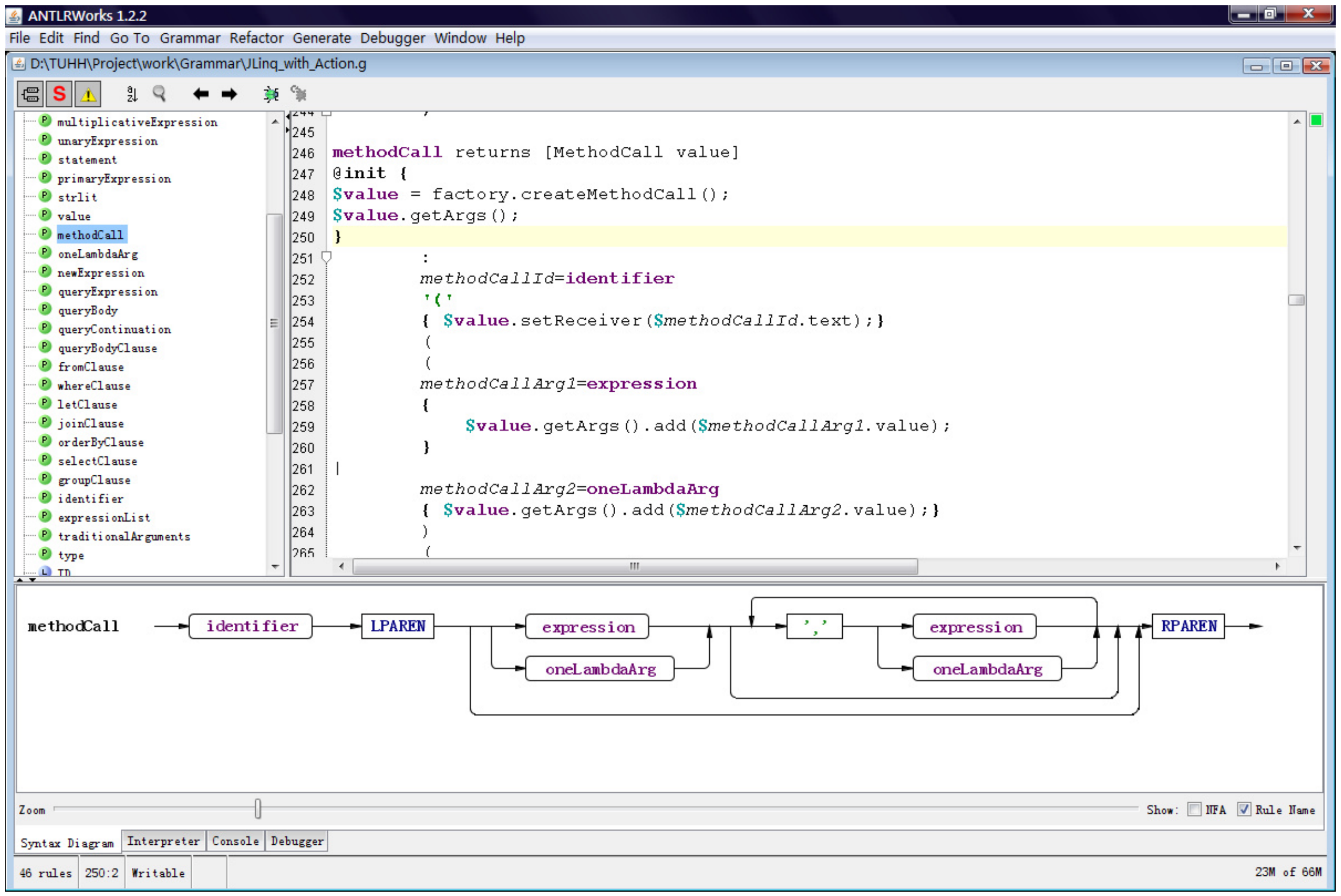


Figure 2.1: Grammar diagram of *value*



```
methodCall returns [MethodCall value]
@init {
$value = factory.createMethodCall();
$value.getArgs();
}
        :
        methodCallId=identifier
        '('
        { $value.setReceiver($methodCallId.text);}
        (
        (
        methodCallArg1=expression
        {
            $value.getArgs().add($methodCallArg1.value);
        }

        methodCallArg2=oneLambdaArg
        { $value.getArgs().add($methodCallArg2.value);}
        )
        (
```

Figure 2.2: Screenshot of ANTLRWork

The remaining grammar rules can be roughly divided into two categories: expressions and clauses.

query expression

Clauses are the actual building blocks for a query expression. Every query expression starts with a *from* clause followed by a *query body*. A query body starts optionally with one or more *query body clauses*, followed by a *select* clause or a *group* clause, then followed optionally by a *query continuation*. A query continuation clause, starting with an *into* keyword, allows to store the result of the preceding query that provides the source to the further query. Figure 2.3 and Figure 2.4 show the grammar diagrams of query expression and query body.

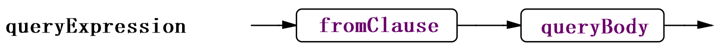


Figure 2.3: Grammar diagram of *query expression*

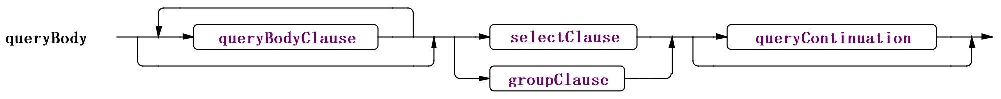


Figure 2.4: Grammar diagram of *query body*

query body

Each query body clause will then match with one or more of the following clauses: *from* clause, *let* clause, *where* clause, *join* clause and *order by* clause. Because arbitrary number of query body clauses can repeat itself in arbitrary order within each query body, sophisticated and complicated queries can be composed and refined easily.

primary expression

Once we have the clauses defined, we need to define the expressions that fill the gaps between them. *Primary expression* is the bridge connecting expressions and query clauses by defining query expression as one of its matching alternative. Furthermore, primary expression also matches *method call*, which represents a method invocation, *new expressions*, which initiates an object, as well as *expression*, recursively. Figure 2.5 shows the grammar diagram of primary expression.

method call

A method call clause can match either a *normal method call* or a *casting method call*. A normal method call invokes a method on its receiver, taking one or more expression or lambda expression as parameter. Besides it can be used to compose query with standard query operation. This clause is vital for the

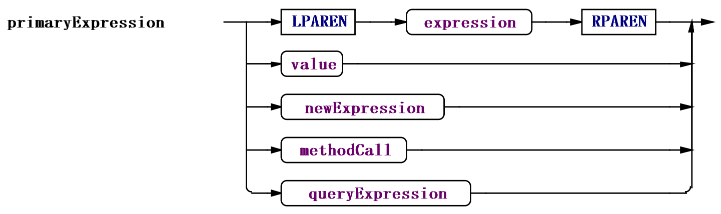


Figure 2.5: Grammar diagram of *primary expression*

next phase, where we need to transform each textual query expression into a sequence of invocations of standard query methods, the same job as the C# compiler does behind the scenes. Figure 2.6 shows the grammar diagram of normal method call.

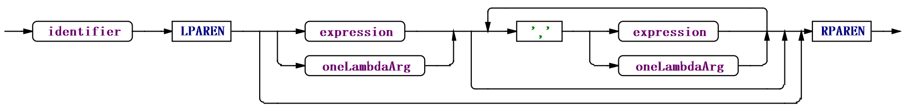


Figure 2.6: Grammar diagram of *normal method call*

A casting method call is used solely in the second phase of a transformation to produce a `Cast<T>()` invocation for *from* clause, *join* clause and *join into* clause with explicit type declaration T. More details about transforming textual query expression into standard query expressions will be described in Chapter 4.

new expression

A new expression instantiates an object in both traditional way where a type name of which the object being instantiated is provided, and in anonymous way where the type name is absent. The latter form of anonymous type is a new feature introduced in C# 3.0 and we also include it in our project. When instantiating with anonymous type, parameters are given within a pair of braces ({and}).

Each other kind of expression rules can match a certain expression according to its specific usage and characteristic. Up to now, the LINQ grammar is reproduced with ANTLR grammar rules, which can then generate a concrete syntax tree in terms of the metamodel described in Chapter 3.

Figure 2.7 on page 11 shows fragment of an interpretation diagram of ANTLRWork on how it parses a simple LINQ query:

```
from c in Customers
from o in Orders
select new {c.name, o.orderID}
```

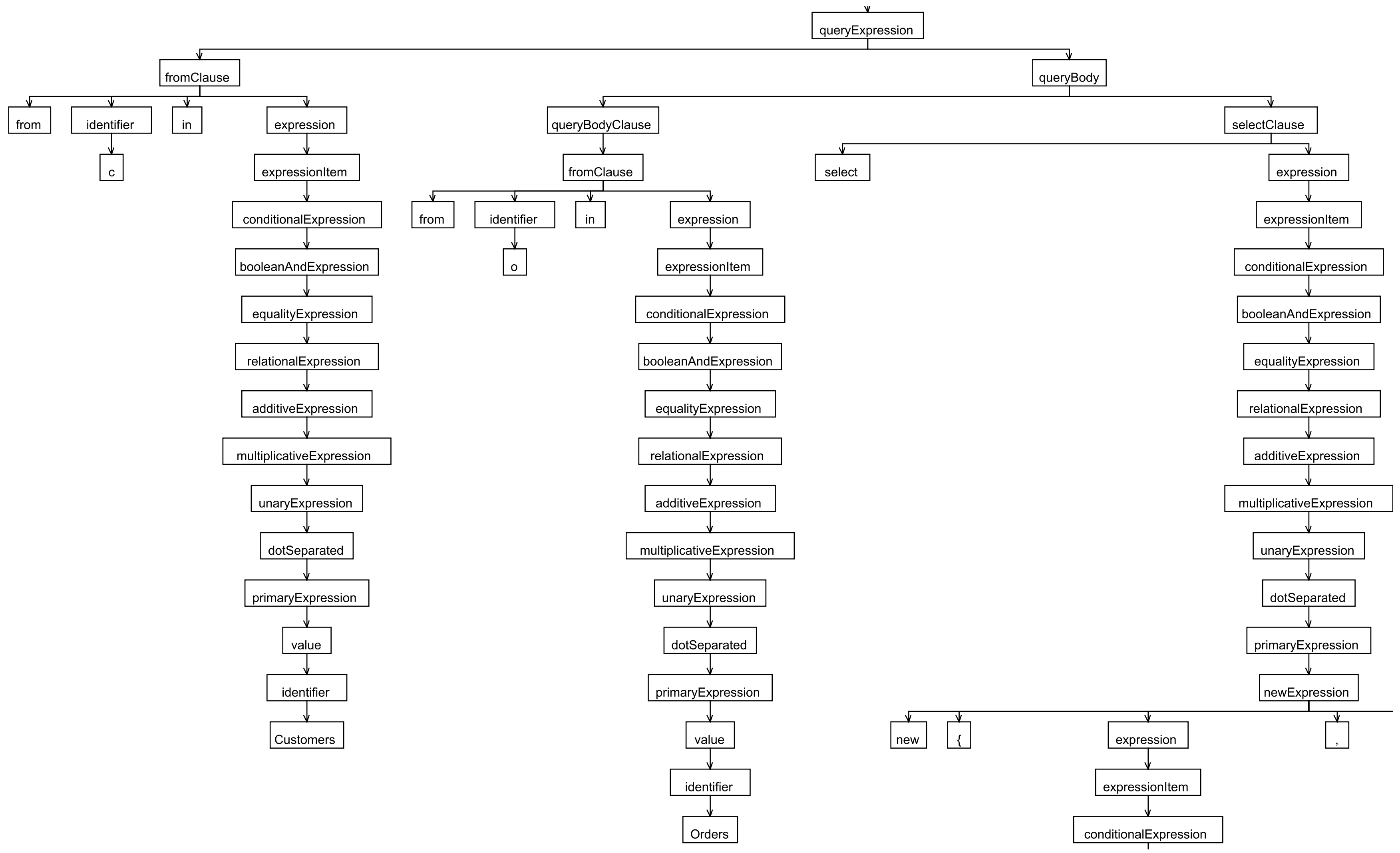


Figure 2.7: Fragment of parsing diagram of *query expression*

ANTLR can then generate the code of a lexer and a parser for the DSL in a specified target language, which is in our project Java. When launching the lexer and parser, the lexer first transfer the input string representing the query into string stream, and consequently into token stream. The token stream is then taken as the input of the parser, which finally produces the concrete syntax tree by matching its root rule. Listing 2.1 gives the code that launches the lexer and parser to generate the concrete syntax tree corresponding to the input string.

```java
public static CSTExpr getCST(String toParse)
{
    JLinq_with_ActionLexer lex = new
        JLinq_with_ActionLexer(new
ANTLRStringStream(toParse));
    CommonTokenStream tokens = new CommonTokenStream(lex)
        ;
    JLinq_with_ActionParser parser = new
        JLinq_with_ActionParser(tokens);
    try
    {
      linqExpression_return parsingRes = parser.
          linqExpression();
      CSTExpr cst = parsingRes.value;
      return cst;
    } catch (RecognitionException e)
      {
          e.printStackTrace();
      }
    return null;
}
```

Listing 2.1: Launch of the lexer and parser

Chapter 3

CST Metamodel

3.1 Class hierarchy of CST metamodel

The CST (Concrete Syntax Tree) metamodel mainly serves the purpose of an intermediate representation of each LINQ query as a tree. With the help of a concrete syntax tree, a query in either textual syntax or method-invocation syntax, composed in whatever manner, as long as it is a legal and syntactically well-formed one, can be represented in a structural and normalized way.

Converting a query into the corresponding concrete syntax tree can be considered a process of normalization. This intermediate step is vital for the next phase when transforming the concrete syntax tree into a sequence of standard query operations (SQO), because that phase requires the application of some transformation rules that take normalized query as their input, and this normalization is a guarantee that the transformation can succeed.

Therefore, the metamodel should be able to well reflect all the logical units and elements of the LINQ grammar, as well as the interplay and relationship between them. In terms of an object-oriented programming language, the former are the interfaces and classes in a specific hierarchy, and the latter are the association and multiplicity between them.

We first define the base interface of all other interfaces: `ExprOrLambdaExpr` that has two direct sub interfaces: `CSTExpr` and `LambaExpr`. This is to differentiate the newly introduced Lambda expression that implements `LambdaExpr`, and all the other kinds of expressions that implement the sub interfaces of `CSTExpr`. Figure 3.1 shows a fragment of the class diagram of the CST metamodel. Since the class diagram is relatively large, Figure 3.1 only shows those classes that represent the grammar element of LINQ. As further constructs are discussed (e.g. Boolean and arithmetic expressions) their corresponding metamodel fragment will also be shown.

Interface `QueryExpr` represents a query expression, which is the entry point of a query expression, and also the root node in a concrete syntax tree when the query is composed in textual syntax.

When the query is composed in method-invocation syntax, the entry point and the root node of the concrete syntax tree will be an instance of interface `DotSeparated`. `DotSeparated` represents the kind of structures that is separated by dots. A typical method invocation looks like:

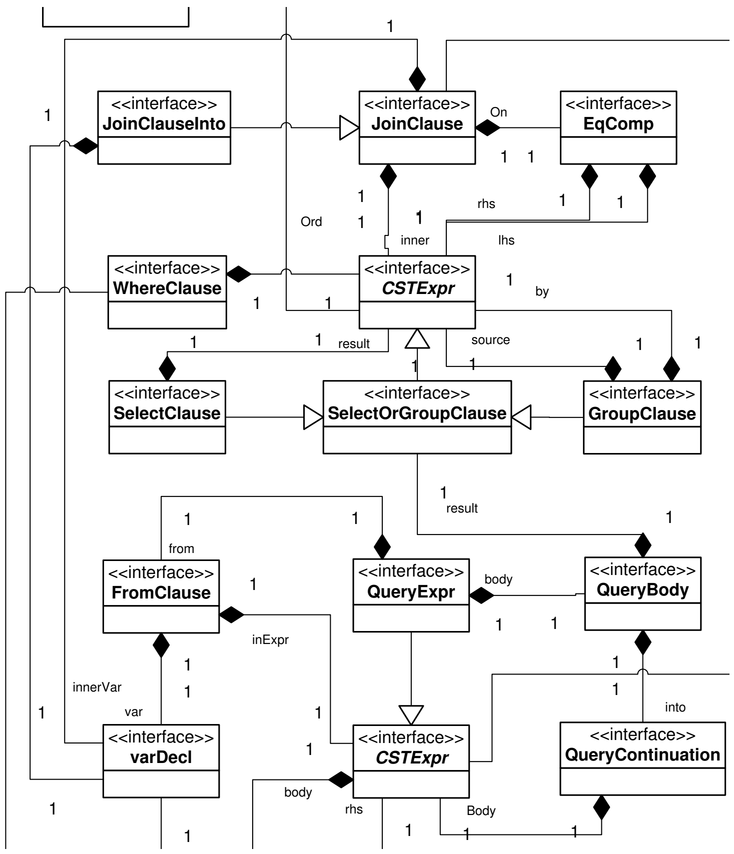


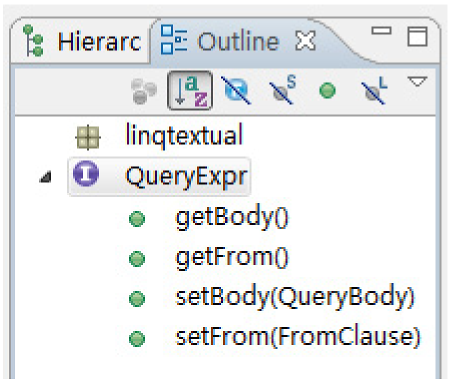
Figure 3.1: Fragment of the class diagram of CST metamodel (clauses)

```
x1.Select(a=>a)
```

where `Select(a=>a)` is an instance of interface `MethodCall` taking the lambda expression `a=>a` as its arguments. The above expression as a whole is an instance of `DotSeparated`, in which the identifier `x1` and the method invoked to `x1` is separated by a dot.

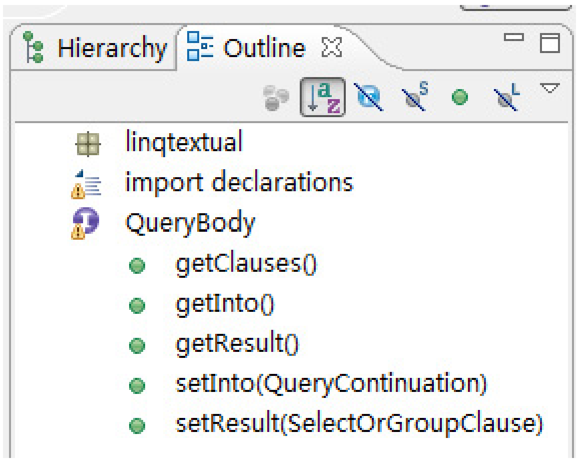
To maintain the association and multiplicity relationship between nodes, each interface defines getters and setters of other interfaces that will be children nodes of it. For those nodes with multiplicity of exactly one a single getter and a setter suffice, while for those with multiplicity of more than one, a list of children nodes will be kept. For example, a node of `QueryExpr` should have two children, each of type `FromClause` and `QueryBody`. Each `QueryBody` then has an optional list of `QueryBodyClause`, a mandatory result expression of type `SelectOrGroupClause`, and an optional `QueryContinuation`. Figure 3.2 shows the outline of `QueryExpr` and Figure 3.3 shows that of `QueryBody`.

There are another group of expressions that are not part of the grammar elements of LINQ, but can be used within these elements. These expressions include for example `UnaryExpr`, `BinaryExpr`, `BoolExpr` and so on. The uses and purposes of these expressions are quite self-explanatory. Figure 3.4 shows the fragment of the class diagram of CST metamodel where these expressions



linqtextual
QueryExpr
getBody()
getFrom()
setBody(QueryBody)
setFrom(FromClause)

Figure 3.2: Outline of interface `QueryExpr`



linqtextual
import declarations
QueryBody
getClauses()
getInto()
getResult()
setInto(QueryContinuation)
setResult(SelectOrGroupClause)

Figure 3.3: Outline of interface `QueryBody`

are located.

Note that not all interfaces in the metamodel are implemented. Some interfaces just have the goal of generalization of some other interfaces, and they therefore do not need to and cannot be implemented. Such interfaces include `PrimaryExpr`, `BinaryExpr`, `QueryBodyClause` and `SelectOrGroupClause`. For example, `SelectOrGroupClause` generalizes both interfaces `SelectClause` and `GroupClause`. In this way, the `setResult()` method in `QueryBody` will take an object of type SelectOrGroupClause as its argument, reflecting the LINQ grammar rule that the result of a query body must be either a select clause or a group clause. `PrimaryExpr` generalizes `Literal`, `MethodCall`, `NewExpr` and `QueryExpr`, `BinaryExpr` generalizes `ArithExpr` (representing arithmetic expression) and `BoolExpr` (representing Boolean expression), while `QueryBody` generalizes `FromClause`, `LetClause`, `JoinClause`, `OrderByClause` and `WhereClause`.

Finally there is a factory class in our metamodel, which can be used to create various types of nodes of the concrete syntax tree in a unified way. This class is being used mainly in the semantic actions to build the nodes, which will be described in Section 2. Moreover, the factory class is also used in the cloning

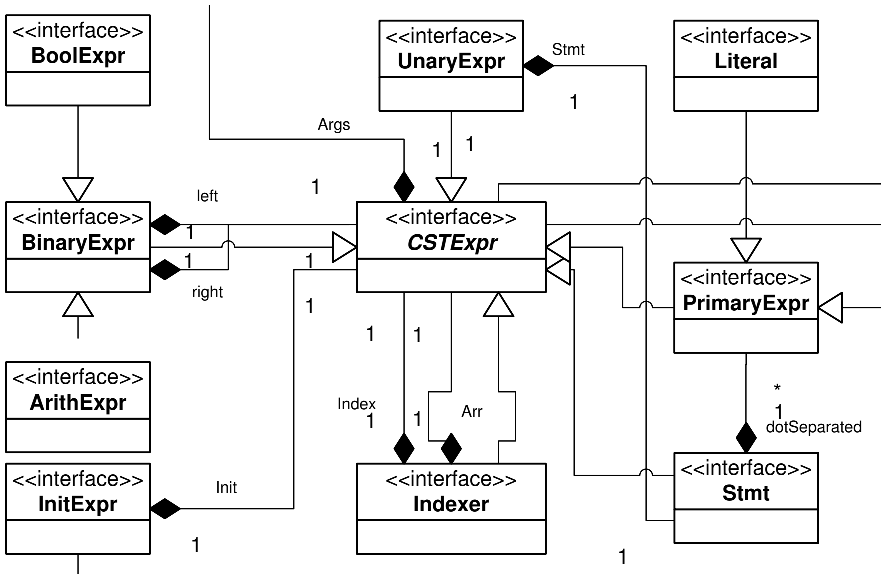


Figure 3.4: Fragment of the class diagram of CST metamodel (expresions)

visitor, which will be described in Section 3, Chapter 4.

3.2 Semantic actions building CST

Now that we have the metamodel that can describe a query as a tree, the next step is to build this concrete syntax tree as the query syntax is being recognized by the ANTLR parser. ANTLR provides a mechanism called semantic actions. With semantic actions, we can specify which code should be executed when a particular grammar rule is matched. These action codes are executable Java statements. In our case therefore, we can place in semantic action blocks the codes that should be executed to build the node of the concrete syntax tree when the recognition of the corresponding grammar rule occurs.

The factory class mentioned in 3.1 is used to instantiate corresponding nodes. Once a node is constructed, its children nodes are also specified with the results matched, by the respective setters.

We will now examine in detail the semantic action of the grammar rule for query body as an example. The other semantic actions are formulated in a similar way. Listing 3.1 shows the grammar for query body with its semantic actions within the curly braces.

```
queryBody returns [QueryBody value]
@init {
$value = factory.createQueryBody();
List<QueryBodyClause> clauses =$value.getClauses();
SelectOrGroupClause selOrGroup = null;
QueryContinuation qc = null;
}
```

```
8          :(qbc=queryBodyClause { clauses.add($qbc.value); }
               )*
9           (queryBodySel=selectClause { selOrGroup =
               $queryBodySel.value; } | queryBodygrp=
               groupClause { selOrGroup = $queryBodygrp.value
               ; })
10           (queryBodyQuCon=queryContinuation { qc =
                $queryBodyQuCon.value; } )?
11         {
12          $value.setResult(selOrGroup);
13          $value.setInto(qc);
14         }
15         ;
```

Listing 3.1: Grammar rule for query Body and its semantic action

We already know that each query body has an optional list of query body clauses, followed by the resulting mandatory group clause or select clause, and then followed by an optional query continuation. At line 3 an object of type `QueryBody` is created by the factory, which will be the return value of this grammar rule. At line 4 the list of query body clauses is returned, so that we can add at line 8 the following query body clauses we will meet. At line 9, either an object of type `SelectClause` or `GroupClause` will be assigned to variable `selOrGroup`, depending on which one is matched. This value is assigned to the result field of the query body through the setter at line 12. At line 13 the query continuation is also assigned regardless of being null or not.

The object value will be returned at the end of the semantic action and will be used in the matching of a query body in another grammar rule. In this way, a complete concrete syntax tree can be constructed according to the input query string.

3.3 Testing using visitors

The concrete syntax tree can also be used to verify that a query is syntactically well-formed, that is, a query is conformant to the LINQ grammar defined in the C# Language Specification. To verify that our parser and our unparser (i.e. to unparse a CST into a string) are actually inverses of each other, the following testing methodology can be followed:

1. parse input string `s1` into concrete syntax tree `e1`
2. unparse `e1` into string `s2`
3. parse string `s2` into concrete syntax tree `e2`
4. compare `e1` and `e2`

In order to implement the above test, we need to get a string representation of a concrete syntax tree in a way that the resulting string should syntactically equal to the original query. Further, since the query expression can have nested queries and other kinds of expressions at different locations in itself, we also need a way to walk through the generated concrete syntax tree recursively.

Visitor design pattern is applied here. The visitor interface contains declarations of methods that will be called before and after visiting each kind of node. A specific implementation of this interface can then determine which of or both of the pre-visiting and post-visiting methods should be overridden.

To actually walk through the concrete syntax tree, a walker class is implemented that can navigate the tree recursively and exhaustively. This class has only one public method `walk` that takes an instance of `ExprOrLambdaExpr`, which is a concrete syntax tree or a sub tree to be walked, and an instance of an implementation of `LinqtextualVisitor`, which is the visitor that is being sent to visit the tree, as its two arguments. In all other private methods that visit different kinds of nodes, algorithm shown in Listing 3.2 is followed.

```
private T VisitTypeA(TypeA tree, Visitor<T> v) {
    If (tree instanceof FirstSubTypeOfA) {
          FirstSubTypeOfA a1 = (FirstSubTypeOfA)tree;
          v.preFirstSubTypeOfA(a1);
          T resFieldA = walk(a1.getFieldA(), v);
          T resFieldB = walk(a1.getFieldB(), v);
          ...
          return v.postFirstSubTypeOfA(a1, resFieldA,
               resFieldB, ... )
    } else if (tree instanceof SecondSubTypeOfA) {
      ...
    }
      ...
}
```

Listing 3.2: Algorithm of walkers of different kinds of nodes

As in Listing 3.2 shown, each node will be visited after all of its children nodes have been visited. The visit result of children nodes will hence be used as the parameters for the visit of current node.

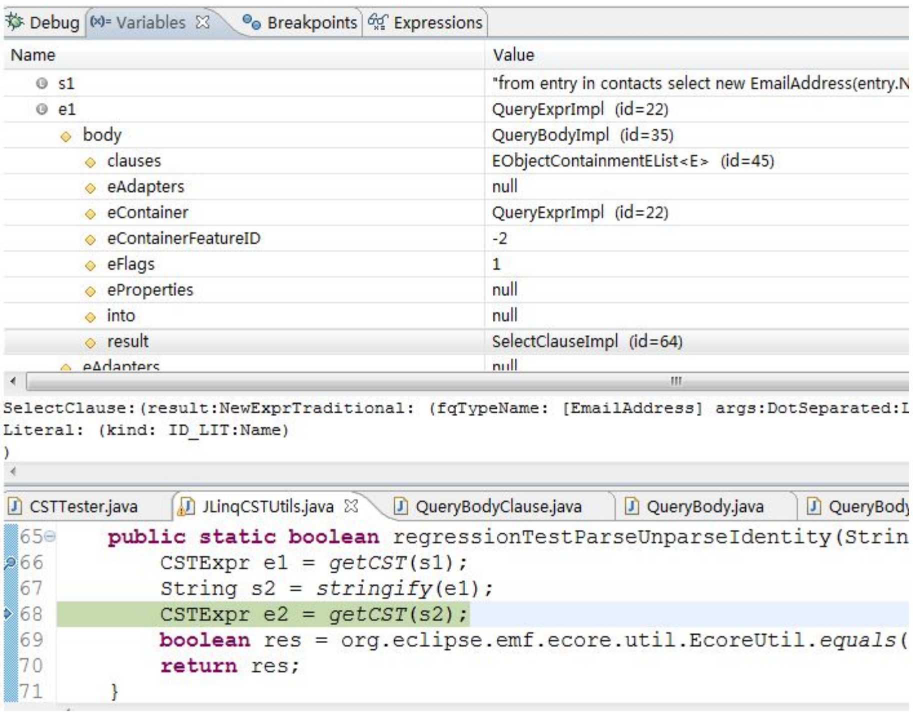
To fulfill the task of parsing a concrete syntax tree into string, a class `LinqtextualToString` is written that implements the visitor interface. In each post-method, the string result is constructed in accordance with the query syntax itself, using the parsing result of its children nodes. Because of this post-order navigating behavior, we only need to overwrite all the post-methods in the interface, and all the pre-methods are kept blank.

Listing 3.3 shows the post-method for visiting a from clause. Recall the algorithm shown in Listing 3.2, we can see that the arguments `resIn` and `resvarDecl` are the results of visiting the two children of a from clause: the inner expression and the variable declaration, which are assigned before the method `postFromClause` is being called, and are referenced in that method to produce the final string result.

```
public String postFromClause(FromClause fc,
 String resIn, String resVarDecl) {
     String res = "from " + resVarDecl + " in " + resIn;
     return res;
}
```

Listing 3.3: Method `postFromClause`

Now that we have the walker and the visitor prepared, we are ready to do the test. Figure 3.5 shows the screenshot of the Debug perspective of Eclipse during a test. The upper window shows that an object of the concrete syntax tree `e1`, which has been parsed from the string query input `s1`, is expanded so that the node `result`, child node of node `body` is shown.



Name	Value
s1	"from entry in contacts select new EmailAddress(entry.N
e1	QueryExprImpl (id=22)
body	QueryBodyImpl (id=35)
clauses	EObjectContainmentEList<E> (id=45)
eAdapters	null
eContainer	QueryExprImpl (id=22)
eContainerFeatureID	-2
eFlags	1
eProperties	null
into	null
result	SelectClauseImpl (id=64)
eAdapters	null

```
SelectClause:(result:NewExprTraditional: (fqTypeName: [EmailAddress] args:DotSeparated:I
Literal: (kind: ID_LIT:Name)
)
```

```
    public static boolean regressionTestParseUnparseIdentity(Strin
        CSTExpr e1 = getCST(s1);
        String s2 = stringify(e1);
        CSTExpr e2 = getCST(s2);
        boolean res = org.eclipse.emf.ecore.util.EcoreUtil.equals(
        return res;
    }
```

Figure 3.5: Screenshot of Debug perspective of Eclipse during a test

Chapter 4

Transformation into SQO

4.1 Necessity and purpose of transformation into SQO

In LINQ, queries can be authored in either textual query expression style or method-call style. In fact, the C# compiler transforms internally all textual query expressions into series of method calls, also called Standard Query Operations (SQO), before performing further process[2]. Once the transformation is finished, the compiler will bind the queries with the actual method bodies.

This transformation is vital and necessary in that it provides considerable flexibility to the query implementers. Specifically, query expressions are translated into invocations of methods named `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast`. These methods are expected to have particular signatures and return types, as described in the query expression pattern. The C# compiler does not care about who provides these methods, or how these methods behave. They can be called as long as they adhere to that pattern. They can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query. Different providers then can have their own definition for these methods, allowing implementation-specific query behavior. Because of this flexibility, different LINQ implementations like LINQ to Objects, DLINQ, and LINQ to Entities can coexist under the same LINQ syntax.

The transformation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. Following transformation of query expressions, the resulting method invocations are type-checked as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

4.2 Transformation rules

There are eighteen rules that define how queries are transformed. A query expression is processed by repeatedly applying these transformation rules until no further reductions are possible. The transformations are listed in order of application and divided into four phases. The start of each phase assumes that the transformations in the preceding phases have been performed exhaustively. Once exhausted, a phase will not later be revisited in the processing of the same query expression. Table 4.1 gives an overview of the query patterns for each transformation rule.

Phase	Rule	Query Expression Pattern		
Phase 1	Rule 1	inline query continuation		
Phase 2	Rule 2	`FromClause`		annotation to cast
	Rule 3	`JoinClause`		
	Rule 4	`JoinIntoClause`		
Phase 3	Rule 5	identity query (`degenereate`)		
Phase4	Rule 6	`FromClause`	`FromClause`	`SelectClause`
	Rule 7			Non-`SelectClause`
	Rule 8		`JoinClause`	`SelectClause`
	Rule 9			Non-`SelectClause`
	Rule 10		`JoinIntoClause`	`SelectClause`
	Rule 11			Non-`SelectClause`
	Rule 12		`OrderByClause`	
	Rule 13		`WhereClause`	
	Rule 14		`LetClause`	
	Rule 15		non-identity `SelectClause`	
	Rule 16		identity `SelectClause`	
	Rule 17		non-identity `GroupbyClause`	
	Rule 18		identity `GroupbyClause`	

Table 4.1: Overview of transformation rules

Each transformation rule is listed in Table 4.2 to Table 4.19, specifying the patterns of the queries before and after the transformation is applied, the precondition of the transformation, and the transformation algorithm.

Transformation Rule 7, 9, 11 and 14 introduce range variables with *transparent identifiers* denoted by *. More details about transparent identifiers are discussed in Section 4.

The first phase comprises only Transformation Rule 1, which removes inline query continuation. It will be applied repeatedly until there is no query continuation in any subtree of the resulting tree.

pattern	`  from x1 in e1 ...  into x2 ...`
	`→ from x2 in (from x1 in e1 ...)  ...`

precondition	`function T1 ( q : QueryExpr ) : QueryExpr`
	`when q.body.into != null`

algorithm	
	```
	new QueryExpr {
	  from = new FromClause{
	        var = q.body.into.var,
	        in = new QueryExpr {
	            from = q.from,
	            body = new QueryBody {
	                clauses = q.body.clauses,
	                result = q.body.result
	              }
	       }
	   }
	   body = q.body.into.body
	}
	```

Table 4.2: Transformation Rule 1

The second phase comprises Transformation Rule 2 to Rule 4, which reformulate all explicit type annotations in terms of casting method calls. It will be applied repeatedly until there is no explicit type annotation in any `FromClause`, `JoinClause` or `JoinIntoClause` in any subtree of the resulting tree.

pattern	`  from T x1 in e1`
	`→ from x1 in e1.Cast<T>()`

precondition	`function T2 ( q : QueryExpr ) : QueryExpr`
	`when q.from.var.type != null`

algorithm	
	```
	new QueryExpr {
	    from = new FromClause{
	           var = q.from.var,
	                in = q.from.in
	         }
	    body = q.qbody.Cast<q.from.var.type>()
	}
	```

Table 4.3: Transformation Rule 2

pattern	`   join T x1 in e1 on k1 equals k2` → `join x1 in e1.Cast<T>() on k1 equals k2`

precondition	`function T3 ( j : JoinClause ) : JoinClause`
	`when j.innerVar.type != null`

algorithm	`new JoinClause {`
	` innerVar = j.innerVar,`
	` inner = j.inner.Cast<j.type>(),`
	` on = j.on`
	`}`

Table 4.4: Transformation Rule 3

pattern	`   join T x1 in e1 on k1 equals k2 into g` → `join x1 in e1.Cast<T>() on k1 equals k2 into g`

precondition	`function T4 ( ji : JoinClauseInto) : JoinClauseInto`
	`when ji.innenVar.type != null`

algorithm	`new JoinClauseInto{`
	` innenVar = ji.innerVar,`
	` inner = ji.inner.Cast<ji.type>(),`
	` on = ji.on,`
	` resultVar = ji.resultVar`
	`}`

Table 4.5: Transformation Rule 4

The third phase comprises Transformation Rule 5 to rewrite degenerate query. According to the C# Language Specification, A degenerate query expression is "*one that trivially selects the elements of the source*". A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. "*It is important however to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling Select on the source[1].*"

pattern	`from x1 in e1 select x1` → `e1.Select(x1 => x1)`

precondition	`function T5 ( q : QueryExpr) : QueryExpr`
	`when q.from.var == q.qbody.result.result`

algorithm	`q.from.in.Select( (q.from.var) => (q.from.var) )`

Table 4.6: Transformation Rule 5

The fourth phase comprises Transformation Rule 6 to Rule 18 which iteratively reduce the query clauses until none of their preconditions holds for any subtree of the resulting standard query operation query. This phase performs the actual reduction on the query expression.

pattern	`from x1 in e1 from x2 in e2 select e3` → `e1.SelectMany(x1 => e2, (x1, x2) => e3)`

precondition	`function T6 ( q : QueryExpr) : QueryExp`
	`when q.body.clauses[0] instanceof FromClause and`
	`     q.body.clauses.size == 1 and`
	`     q.body.result instanceof SelectClause`

algorithm	`x1 = q.from.var;`
	`x2 = q.body.clauses[0].var;`
	`e2 = q.body.clauses[0].in;`
	`e3 = q.body.result;`
	`q.from.in.SelectMany( x1 = >e2, (x1, x2) => e3 )`

Table 4.7: Transformation Rule 6

pattern	`from x1 in e1 from x2 in e2...` → `from * in e1.SelectMany(x1->e2, (x1, x2)=>new x1, x2)...`

precondition	`function T7 ( q : QueryExpr) : QueryExpr`
	`when q.body.clauses[0] instanceof FromClause`
	`and (q.body.clauses.size()==1 and`
	`q.body.result instanceof  GroupbyClause)  or`
	`       q.body.clauses.size()>1 )`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`x2 = q.body.clauses[0].var;`
	`e2 = q.body.clauses[0].in;`
	`transId = newTransId(q);`
	`QueryExpr newIn=e1.SelectMany(x1=>e2,(x1,x2)=>new {x1,x2});`
	`newfrom = new FromClause {`
	`             var = transId,`
	`             in = newIn`
	`}`
	`QueryExpr result = new QueryExpr {`
	`from = newFrom,`
	`body = q.body`
	`}`

Table 4.8: Transformation Rule 7

pattern	`from x1 in e1join x2 in e2 on k1 equals k2`
	`select e3`
	→ `e1.Join(e2, x1=>k1, x2=>k2, (x1, x2)=>e3)`
precondition	
	`function T8 ( q : QueryExpr) : QueryExpr`
	`when q.body.bclauses = JoinClause`
	`and`
	`q.body.clauses.size()==1`
	`and`
	`q.body.result instanceof SelectClause`
algorithm	
	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`x2 = q.body.qbclauses.join.innervar;`
	`e2 = q.body.qbclauses.join.innerexp;`
	`k1 = q.body.clauses[0].on.lhs;`
	`k2 = q.body.clauses[0].on.rhs;`
	`e3 = q.body.result;`
	`e1.Join( e2, x1=>k1, x2=>k2, (x1, x2)=>e3 )`

Table 4.9: Transformation Rule 8

pattern	`   from x1 in e1 join x2 in e2 on k1 equals k2...`
	`→from * in e1.Join(e2,x1=>k1,x2=>k2,(x1,x2)=>new x1, x2)`

precondition	`function T9 ( q : QueryExpr) : QueryExpr`
	`when q.body.clauses[0] instanceof JoinClause`
	`and (q.body.clauses.size()==1 and`
	`q.body.result instanceof  GroupbyClause)  or`
	`       q.body.clauses.size()>1 )`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`x2 = q.body.clauses[0].innerVar;`
	`e2 = q.body.clauses[0].inner;`
	`k1 = q.body.clauses[0].on.lhs;`
	`k2 = q.body.clauses[0].on.rhs;`
	`transId = newTransId(q);`
	`QueryExpr newIn = e1.Join(e2, x1=>k1, x2=>k2,`
	`                         (x1,x2) => new{x1,x2} );`
	`newFrom = new FromClause {`
	`             var = transId,`
	`             in = newIn`
	`}`
	`QueryExpr result = new QueryExpr {`
	`from = newFrom,`
	`body = q.body`
	`}`

Table 4.10: Transformation Rule 9

pattern	`from x1 in e1 join x2 in e2 on k1 equals k2 into g`
	`select e3`
	→ `e1.GroupJoin(e2, x1=>k1, x2=>k2, (x1, g)=>e3)`

precondition	`function T10 ( q : QueryExpr) : QueryExpr`
	`when q.body.bclauses = JoinClauseInto`
	`and`
	`q.body.clauses.size()==1`
	`and`
	`q.body.result instanceof SelectClause`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`x2 = q.body.clauses[0].innerVar;`
	`e2 = q.body.clauses[0].inner;`
	`k1 = q.body.clauses[0].on.rhs;`
	`k2 = q.body.clauses[0].on.lhs;`
	`g = q.body.clauses[0].resultVar`
	`e3 = q.body.result;`
	`e1.GroupJoin( e2, x1=>k1, x2=>k2, (x1, g)=>e3 )`

Table 4.11: Transformation Rule 10

pattern	`from x1 in e1 join x2 in e2 on k1 equals k into g...`
	`→from * in e1.GroupJoin(e2, x1=>k1, x2=>k2,`
	`(x1,g)=>new x1, g )...`

precondition	`function T11 ( q : QueryExpr) : QueryExpr`
	`when q.body.clauses[0] instanceof JoinClauseInto`
	`and (q.body.clauses.size()==1 and`
	`q.body.result instanceof  GroupbyClause)  or`
	`       q.body.clauses.size()>1 )`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`x2 = q.body.clauses[0].innerVar;`
	`e2 = q.body.clauses[0].inner;`
	`k1 = q.body.clauses[0].on.lhs;`
	`k2 = q.body.clauses[0].on.rhs;`
	`g = q.body.clauses[0].resultVar;`
	`transId = newTransId(q);`
	`QueryExpr newIn = e1.Join(e2, x1=>k1, x2=>k2,`
	`                          (x1,g) => new{x1,g } );`
	`newFrom = new FromClause {`
	`             var = transId,`
	`             in = newIn`
	`}`
	`QueryExpr result = new QueryExpr {`
	`from = newFrom,`
	`body = q.body`
	`}`

Table 4.12: Transformation Rule 11

pattern	`from x1 in e1 orderby k1, k2, k3...` → `from x1 in e1. OrderBy(x1=>k1). ThenBy(x1=>k2).ThenBy (x1=>k3)...`

precondition	`function T12 ( q : QueryExpr) : QueryExpr`
	`when q. body.clauses[0] instanceof OrderByClause`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`k1 = q.body.clauses[0].orderings[0];`
	`QueryExpr temp = e1.OrderBy(x1 => k1);`
	`for (int i=1, i < (q.body.clauses.orderings. length), i++){`
	`  temp.ThenBy(x1 => (q.body.clauses[i]. orderings[i]) );`
	`}`
	`new QueryExpr {`
	`  from = new FromClause {`
	`       var = q.from.var;`
	`       in = temp;`
	`}`
	`  body = q.body;`
	`}`

Table 4.13: Transformation Rule 12

pattern	`from x1 in e1 where e2...` → `from x1 in e1.Wher(x1=>e2)...`

precondition	`function T13 ( q : QueryExpr) : QueryExpr`
	`when q.body.clauses[0] = WhereClause`

algorithm	`MethodCall temp = new MethodCall {`
	`  q.from.in.Where( (q.from.var)=>`
	`(q.body.Clauses[0].booltest) ;`
	`new QueryExpr {`
	`  from = new FromExpr {`
	`     var = q.from.var;`
	`     in = temp;`
	`  }`
	`  qbody = q.qbody;`
	`}`

Table 4.14: Transformation Rule 13

pattern	`from x1 in e1 let x2 = e2...`
	→ `from * in e1.Select(x1=>new  x1, x2=>e2)...`
precondition	
	`function T14 ( q : QueryExpr) : QueryExpr`
	`when q.body.clauses[0] = Let`
algorithm	
	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`x2 = q.body.clauses[0].lhs;`
	`e2 = q.body.clauses[0].rhs;`
	`transId = newTransId(q);`
	`QueryExpr newIn = e1.Select (x1=>new{x1,x2=e2});`
	`newFrom = new FromClause {`
	`             var = transId,`
	`             in = newIn`
	`}`
	`QueryExpr result = new QueryExpr {`
	`from = newFrom,`
	`body = q.body`
	`}`

Table 4.15: Transformation Rule 14

pattern	`from x1 in e1 select e2`
	→ `e1.Select(x1=>e2)`
precondition	
	`when q.body.clauses.size() == 0 and`
	`      q.body.result instanceof SelectClause and`
	`q.from.var != q. body.result.result`
algorithm	
	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`e2 = q.body.result.result;`
	`e1.Select( x1 => e2)`

Table 4.16: Transformation Rule 15

pattern	`from x1 in e1 select e2` → `e1`

precondition	`when q.body.clauses.size() == 0 and`
	`      q.body.result instanceof SelectClause and`
	`q.from.var == q. body.result.result`

algorithm	`e1`

Table 4.17: Transformation Rule 16

pattern	`from x1 in e1 group e2 by e3` → `e1.GroupBy(x1=>e3, x1=>e2)`

precondition	`when q.body.clauses.size() == 0 and`
	`      q.body.result instanceof GroupbyClause and`
	`      q.from.var != q. body.result.result.source`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`e2 = q.body.result.source;`
	`e3 = q.body.result.by;`
	`e1.GroupBy(x1=>e3, x1=>e2)`

Table 4.18: Transformation Rule 17

pattern	`from x1 in e1 group x1 by e3` → `e1.GroupBy(x1=>e2)`

precondition	`when q.body.clauses.size() == 0 and`
	`      q.body.result instanceof GroupbyClause and`
	`q.from.var == q. body.result.result.source`

algorithm	`x1 = q.from.var;`
	`e1 = q.from.in;`
	`e2 = q.body.result.source;`
	`e1.GroupBy(x1=>e2)`

Table 4.19: Transformation Rule 18

Through the application of Rule 6 to Rule 18 repeatedly and recursively, a query in textual syntax can always be reduced completely to a series of SQO method calls. This is guaranteed because of the following:

1. After the application of Rule 1, there exists no query continuation in any sub query.
2. A legal query expression always starts with an initial from clause. Because of 1, a legal query expression can now only end with either a select clause or a groupby clause.
3. Any other optional query body clauses are located between the initial from clause and the ending select/groupby clause. These clauses include from clause, join clause, join-into clause, order-by clause, where clause and let clause.
4. Among those optional clauses mentioned in 3 (i.e. those located after the mandatory initial from clause), Rule 6 or Rule 7 applies to the first of them if that is a from clause. In either case, this from clause is removed, reducing the total number of clauses by 1.
5. Cases are similar when the first of them are clauses other than from clause. Rule 8 or Rule 9 reduces a join clause, Rule 10 or Rule 11 reduces a join-into clause, Rule 12 reduces an orderby clause, Rule 13 reduces a where clause, and Rule 14 reduces a let clause. Therefore, all these optional clauses can be eventually reduced, leaving only an initial mandatory from clause and the ending mandatory select/groupby clause.
6. Rule 15 or Rule 16 rewrites a query with only initial from clause and ending select clause into a method call. So does Rule 17 or Rule 18 when the ending clause is a groupby clause.

Consider the following example query:

```
from x in foo where x>0 from y in bar select x+y
```

To transform this query, we first see if Rule 1 to Rule 5 apply. Since it has no query continuation, no explicit type annotation and is not a degenerate query, none of these rules applies. Entering phase 4, the first applicable rule is Rule 13, which reduces the where clause and results in the following:

```
from x in foo.Where(x>0) from y in bar select x + y
```

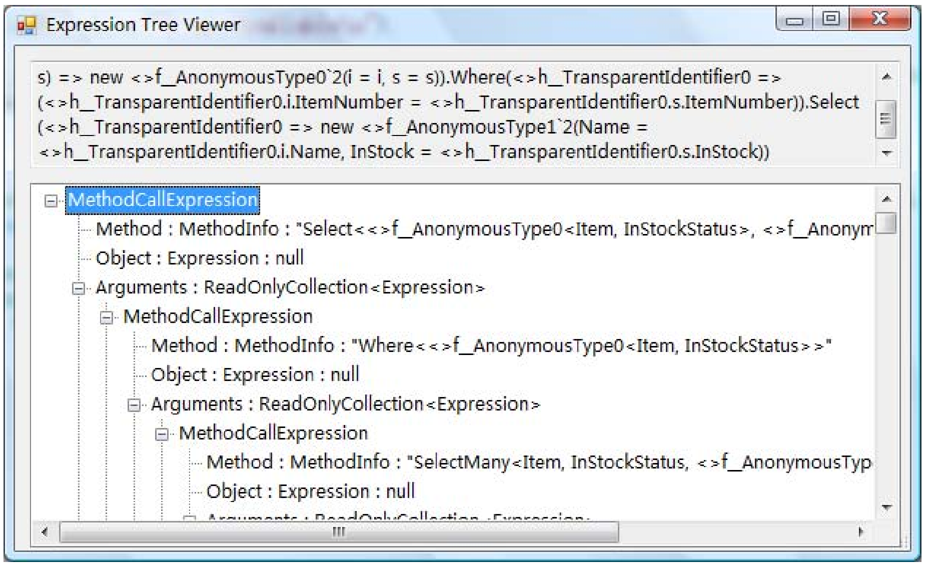
Repeating phase 4, the next applicable rule is Rule 6, reducing the second from clause and resulting in the following:

```
foo.Where(x>0).SelectMany( x=>bar, (x,y)=>x + y )
```

Up to now, we have only a series of method calls and not any query clause. The transformation can stop here and the above is the final result.

It is not easy to "manually" check whether a transformation is correct. Fortunately, there is a LINQ example project called "expression tree viewer" that comes with Visual Studio 2008[7]. This tool originally serves the purpose of being an example of LINQ queries and can represent an expression as a tree, but it can also be a plug-in to the Visual Studio Debugger. Because the C# compiler always transforms any query into Standard Query Operations, and this tool also shows the result of this transformation, we can make use of it by giving a textual LINQ query as input and compare the result with ours. Figure 4.1 shows the expression tree viewer using the following query as its input:

```
from i in items.AsQueryable() from s in statuses.AsQueryable()
where i.ItemNumber==s.ItemNumber select new
{i.Name,s.InStock}
```



```
s) => new <>f_AnonymousType0`2(i = i, s = s)).Where(<>h_TransparentIdentifier0 =>
(<>h_TransparentIdentifier0.i.ItemNumber = <>h_TransparentIdentifier0.s.ItemNumber)).Select
(<>h_TransparentIdentifier0 => new <>f_AnonymousType1`2(Name =
<>h_TransparentIdentifier0.i.Name, InStock = <>h_TransparentIdentifier0.s.InStock))
```

```
MethodCallExpression
    Method : MethodInfo : "Select<<>f_AnonymousType0<Item, InStockStatus>, <>f_Anonym
    Object : Expression : null
    Arguments : ReadOnlyCollection<Expression>
        MethodCallExpression
            Method : MethodInfo : "Where<<>f_AnonymousType0<Item, InStockStatus>>"
            Object : Expression : null
            Arguments : ReadOnlyCollection<Expression>
                MethodCallExpression
                    Method : MethodInfo : "SelectMany<Item, InStockStatus, <>f_AnonymousTyp
                    Object : Expression : null
```

Figure 4.1: Screen shot of expression tree viewer

The upper window shows the result of the transformation, and the lower window shows the tree representation of the resulting SQO query.

4.3 Implementation of the transformation with visitors

The transformation rules introduced above are applicable to query expressions. When demonstrating them, variables like e1 are used to denote source collections within query expressions. This notation is just a simplification and should not be misleading, because such expressions can in turn be query expressions, which also require the application of transformation rules in order that a transformed query does not contain any textual query expression in any subtree.

Because of this fact that a query should be transformed recursively, visitor design pattern is applied again. This time we implement the visitor interface with a so-called “cloning visitor”. *Cloning visitor* returns a duplicate object of the node being visited. This duplication copies not only the node itself, but also all of its children nodes. We use a cloning visitor here because the algorithms of the transformation rules involve changing the shape of a concrete syntax tree by moving some particular sub trees from one place to another. Most of the other sub trees need to be copied to their new place as is. What’s more, using a cloning visitor reduces the chance of having reference issues when manipulating a tree (i.e. portions of a new tree pointing to subtrees in the old one).

The cloning visitor implements all the post-visiting methods of the visitor interface. When visiting each node, a new object is created by the factory class, and all of its field will be then assigned to results of the visit to those nodes.

Each transformation rule can be considered a special case of the cloning visitor, in that when visiting one node, it returns the resulting node after the transformation, instead of simply a copy of it. Hence, each transformation visitor subclasses the cloning visitor, and each only needs to override the post methods for the nodes where the transformation rule should apply. In the overriding method, the transformation is performed once the precondition is fulfilled. For example, the visitor for Transformation Rule 1 overrides method `PostQueryExpr`, the visitor for Transformation Rule 2 overrides method `PostFromClause`, the visitor for Transformation Rule 3 overrides method `PostJoinClause` and the visitor for Transformation Rule 4 overrides method `PostJoinClauseInto`.

4.4 Transparent identifiers

Transformation Rules 7, 9, 11 and 14 introduce transparent identifiers to their outputs. Each transparent identifier has exactly one associated anonymous type, which is introduced during the same rewrite step as the transparent identifier. Consider the following example[1]. We now want to transform the following query into standard query operators.

```
from x in foo let y=f(x) let z=g(x,y) select (x,y,z)
```

Applying Rule 14 once will result in the following intermediate query:

```
from * in foo.Select(x => new { x, y = f(x) }) let z = g(x, y)
select h(x, y, z)
```

[1]Yet Another Language Geek, *http://blogs.msdn.com/wesdyer/archive/2006/12/22/transparent-identifiers.aspx*

Here a transparent identifier is introduced, associating with the anonymous type `new { x, y = f(x)}`. Applying Rule 14 again we will get

```
from * in foo
.Select(x => new { x, y = f(x) }) .Select(* => new { *, z = g(x, y)
}) select h(x, y, z)
```

Again a new transparent identifier is introduced, associating with the anonymous type `new {*, z = g(x, y)}`, in which the * denotes the first transparent identifier introduced earlier. To distinguish the two transparent identifiers, we denote them as `*1` and `*2`. Now we know that `*1` binds to new anonymous type `{ x, y = f(x)}`, while `*2` binds to new anonymous type `{ *1, z = g(x, y)}`. Rewrite the result above with Rule 15 and we get this result:

```
foo.Select(x =>new{x,y=f(x)}).Select(*1=>new{*1,z=g(x, y)})
.Select(*2 => h(x, y, z))
```

Next step we need to resolve the transparent identifiers. When a transparent identifier is in scope, its members are also in scope. For the above result, `*2` is in scope so that `*1` and `z` are in scope. Therefore we rewrite `z` as `*2.z`, and `*1` as `*2.*1`. Moreover, `*1` is in scope so that `x` and `y` are in scope, so we can in turn rewrite `x` as `*1.x` and `y` as `*1.y`. Since we have rewritten `*1` as `*2.*1`, all the occurrences of members of `*1` should also be rewritten. Finally we rewrite `x` as `*1.x` then as `*2.*1.x`. The same applies to `*y` to get `*2.*1.y`. Replacing `*1` with name `transId_1` and `*2` with name `transId_2`, we get the following final result:

```
foo.Select(x => new { x, y = f(x) })
.Select(transId_1 => new
  {transId_1, z = g(transId_1.x, transId_1.y) })
.Select(transId_2=>
  h(transId_2.transId_1.x, transId_2.transId_1.y, transId_2.z))
```

Using the term "scope" above, it looks like as if an environment should be maintained to keep track of the valid scope of each transparent identifier. Using environment, different transparent identifiers can have the same name, as long as they are located in different environment. Certain programming languages like Pascal have the similar situation where different variables with the same name can coexist as long as they are declared in different blocks. While this feature does provide some level of flexibility, it also introduces some ambiguity and compromises readability of the source code. Therefore, variables with the same name, as long as they are all visible within a certain scope, are not allowed by the C# Compiler. This rule also applies to LINQ query. When processing transparent identifier, this means that we do not need to maintain an environment. Without an environment, all identifiers declared in a query, including transparent identifier, must have unique name within the whole query.

We first define a class that represents one transparent identifier.

```
public class TransId extends VarDeclImpl {
        private int index;
        private List list;
        public TransId(String prefix, int ind) {
            index = ind;
            this.setName(prefix + ind);
            list = new ArrayList();
    }
    public int getIndex(){return index;}
    public List getList(){return list;}
    public TransId addParam(Object value) {
            if (value instanceof TransId || value
                instanceof VarDecl)
                    list.add(value);
            return this;
    }
}
```

Listing 4.1: Class `TransId`

Listing 3.4 shows the class declaration of `TransId`. Each object of this class has its prefix and index combined as its name, and a list keeping all the variables in its scope. In the example above, `transId_1` has `x` and `y` in its list, and `transId_2` has `transId_1` and `z` in its list.

Each object of `QueryExpr` has a field of type `TransIdList`, which has a list of all the `TransId` occurring within the query expression and the prefix to be used for all these transparent identifiers.

Having these data structures in hand, we can now have the general algorithm to process transparent identifiers:

1. Before applying any transformation, find a string that is "safe" to be the prefix of any transparent identifier within this query.

2. Apply the transformations rules in order.

3. When Rule 7, Rule 9, Rule 11 or Rule 14 is applied, a new transparent identifier is created, assigned a unique name and added to the list. All its members are also added to the list of that transparent identifier.

4. Transparent identifiers are resolved at the end of the application of this transformation rule.

Two points should be clarified: how to find the "safe" string in step 1, and how to resolve the transparent identifiers in step 4.

A variable name is considered safe if its name does not conflict with any other variable names within the same query. A "fresh" name is therefore safe if its length exceeds the length of the longest variable name in the query. We name the transparent identifier with a prefix that is safe, followed by an index, so that each transparent identifier has a unique name.

To find out the longest variable name, we use a visitor to visit all the variable occurrences within the syntax tree of a query. An initial prefix for a query is set to "`TransparentID_`".Whenever it visits a variable, it compares the length of the name of this variable with the length of the current prefix. When the former is longer, it updates the current prefix by concatenating another string "`TransparentID_`" to the end of the current prefix. At the end of the entire visit, the prefix is the longest one among all the variables in the query.

Another question is how to resolve the transparent identifier. At the end of each transformation rules that can introduce transparent identifier, the resolution takes place. Another visitor is sent to the transformed query expressions, and checks if a variable is the member of any transparent identifier in the list. If it is, rewrite the variable with the name of its parent (preceding) transparent identifier. For example, rewrite `x` with `TransparentID_0.x`. In terms of the CST metamodel, a node of type `DotSeparated` replaces the original variable node. The transparent identifier is also checked in the same way, so that in this example it results in `TransparentID_1.TransparentID_0.x`. This process is repeated until no single variable or no variable that is in front of the any `DotSeparated` can be found in the list. Figure 4.2 shows the resulting subtree of the above example.

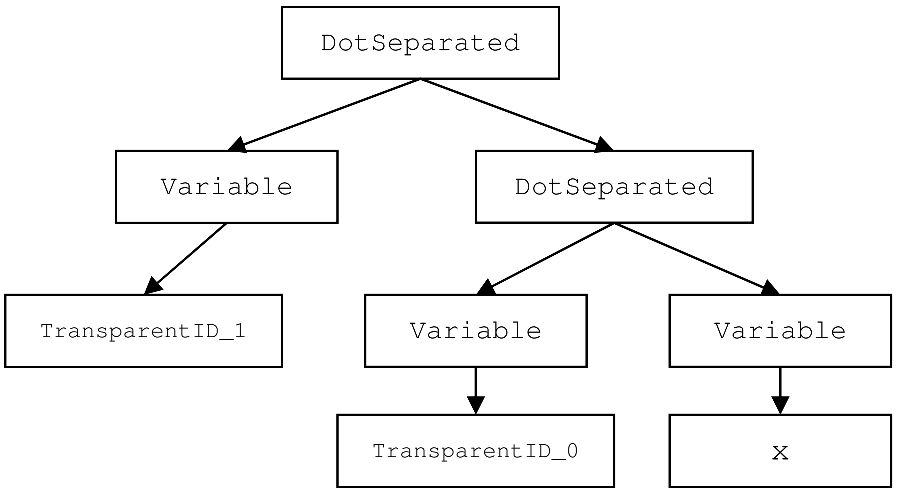


Figure 4.2: Resulting subtree after resolution of transparent identifier

After the transparent identifiers are resolved, the transformation can continue with the following transformation rules. Finally, all textual query expressions can be transformed into series of method calls.

Chapter 5

Conclusions

There is no doubt that LINQ represents an important improvement in the way we write code. This is also one of the reasons why we choose LINQ as the data query mechanism that we were trying to embed into Java. Because of the time limit of the project work, we have to stop the implementation at the point of finishing the transformation into SQO.

A larger use case consists in the situations where, once we have transformed the query into SQO, we have to make sure that it is sent to a database for evaluation. This can be achieved by implementing a plug-in for the Java Compiler so that it can expand embedded queries. Additionally, implementations for the eleven SQO methods (`Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy` and `Cast`) would also be needed. Depending on the source collections (such as Java array, Java Collection and XML), the implementation should take care of accessing that data in its native format. Other data models will likely have different implementations for these methods, and this work should be up to the data model providers. The prototype developed in this project (including parsing and transforming any LINQ query into SQO) is a pre-requisite for the more advanced evaluation functionality. The work reported here can be generally used without the user's concerning about the particular implementations.

We can also learn from the project that, we can apply similar techniques to embed in Java other query languages, such as JPQL (Java Persistence Query Language) or XQuery. The reported technique can thus be summarized (for reuse) in terms of the following steps:

1. Use ANTLR or other similar language recognizing tools to generate a language parser that can recognize the query syntax of the embedded query language.

2. Construct a CST metamodel in Java according to the grammar structure of that query mechanism.

3. Use semantic actions in ANTLR or similar way in other parser generators to build the CST corresponding to a query in that language.

4. Apply some mechanism-specific process if necessary. (In terms of our project, this step involves the transformation into SQO, as this is a LINQ-specific process).

5. Construct a plug-in for Java compiler that can handle the query in Java. Making use of the *Annotation* introduced in Java 5 and JSR 269, the *Pluggable Annotation Processing API*, a customized annotation processor can be plugged-in to the compilation in order to process query[4]. Moreover, further issues such as semantic analysis and type-checking should be considered when implementing this[3].

Bibliography

[1] C# Language Specification 3.0. http://msdn.microsoft.com/en-us/library/bb308966.aspx, March 2007.

[2] The .NET Standard Query Operators. http://msdn.microsoft.com/en-us/library/bb394939.aspx, February 2007.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[4] David Erni and Adrian Kuhn. The Hacker's Guide to Javac. http://www.iam.unibe.ch/ scg/Archive/Projects/Erni08b.pdf, March 2008.

[5] Miguel Garcia and Rakesh Prithiviraj. Rethinking the Architecture of O/R Mapping for EMF in terms of LINQ. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2008, Stuttgart, Germany*, 2008. `http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2008/ese/linq4emf.pdf`.

[6] Paolo Pialorsi and Marco Russo. *Programming Microsoft LINQ*. Microsoft Press, 2008.

[7] Rakesh Prithiviraj. IDE Customization to Support Language Embeddings. Master's thesis, Hamburg University of Technology (TUHH), 2008.

[8] Herbert Schildt. *C# 3.0: A Beginner's Guide*. McGraw-Hill, 2009.