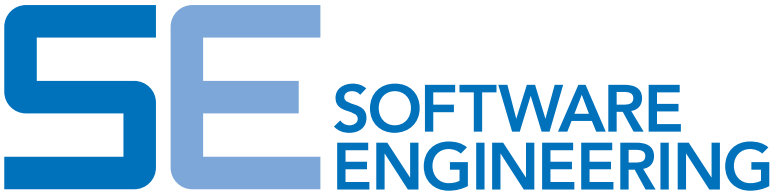
Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering



Diplomarbeit

Typsichere Integration von Abfragesprachen in Scala
Type Safe Integration of Query Languages into Scala

Jan Christopher Vogt
Matrikel-Nr.: 247624

Aufgabenstellung und Erstgutachter: Prof. Dr. Bernhard Rumpe
Zweitgutachter: Prof. Dr. Martin Odersky
Betreuer: Dipl.-Inform. Ingo Weisemöller

Aachen, den 18. August 2011

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 18. August 2011

_____

Abstract

Interaction with relational databases can be troublesome for software developers. The relational data model does not match the data models of most general purpose programming languages. Embedding database queries as SQL Strings does not provide any compile time guarantees and can lead to run time exceptions or bugs like SQL injection vulnerabilities. This thesis introduces Scala Integrated Query (SIQ), a deep and type safe integration of database queries into Scala. It is similar to Microsoft LINQ and the LINQ-to-SQL provider on .NET, but provides stronger compile time guarantees, translates queries with nested results more efficiently and does not introduce new syntax.

SIQ compiles a subset of Scala and a small relational library into SQL. Database tables, in-memory collections and tuples can be combined for arbitrarily nested results, unlike ScalaQuery, Squeryl or SQL, where results are always flat collections. Using the Ferry encoding allows translating nesting efficiently to one or more SQL queries, a number which only depends on the result type, unlike LINQ-to-SQL where it can depend on the size of involved tables.

The prototype is implemented as a library without any special Scala compiler support. Several language features allow lifting queries to abstract syntax trees at run time, while providing precise static type safety. The thesis describes how the necessary constructs are lifted and how type safety can reflect the database schema and SQL specifics. The use of the Lightweight Modular Staging framework makes the library very modular and allows a high degree of code re-use when adapting type safety to different SQL dialects.

Acknowledgments

Many people supported me during my study years and the time of my thesis. I am very grateful and want to mention some of them.

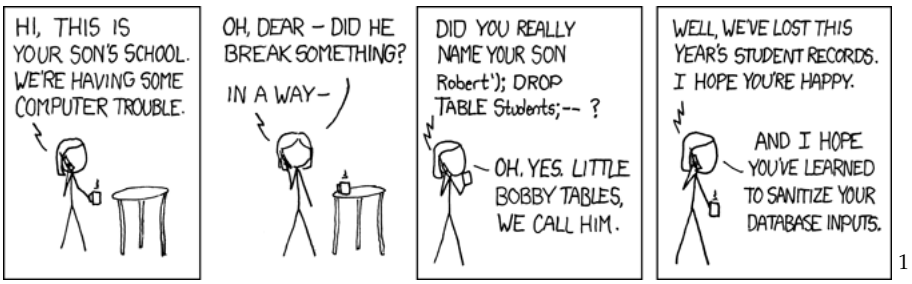
I want to thank Prof. Dr. Bernhard Rumpe for believing in me and supporting my idea of joining the Scala team in Lausanne for my thesis, Prof. Dr. Martin Odersky who welcomed me with open arms and provided me with the perfect working environment for this project, Tiark Rompf and Miguel Garcia for all the reviews, countless useful discussions and challenging feedback that pushed my thesis as far as it went, Tom Schreiber, Phillip Haller and Vlad Ureche for helpful discussions and feedback, Danielle Chamberlain for her cheerful personality and taking care of all the formalities, Markus Look for his organizational support, everybody at LAMP for making my time at EPFL really enjoyable.

Thank you to Mars for her love and amazing spark of happiness every day, to my good friend Florian Landrock for never loosing the spirit and to my good friend Jan Braun for always fun and enlightening discussions about life, computer science and mathematics. Above all, I thank my family, especially my parents, who always believed in me and provided me with a solid foundation in life I could always trust in.

Contents

1. Introduction

Interaction with relational databases can be troublesome for software developers. The relational data model does not match the data models of most general purpose programming languages. Embedding database queries as SQL Strings does not provide any compile time guarantees and can lead to run time exceptions or bugs like SQL injection vulnerabilities.



[1]

Different attempts have been made to overcome these problems. A widely accepted one is Microsoft LINQ[2] on the .NET platform, which integrates queries deeply into .NET programming languages using compiler support. LINQ features nested results and other concepts, which bring the data model of queries closer to the data model of general purpose programming languages. At compile time LINQ lifts a query to an expression tree and at run time a chosen backend provider executes the query within the desired data source. LINQ queries are statically type checked.

In this thesis, we explore a similar approach for Scala we call Scala Integrated Query (SIQ). Unlike LINQ, we do not introduce new syntax. SIQ queries are expressed using ordinary Scala code, like the following example

```
workgroup.map( w => employee.withFilter( _.workgroup_id == w.id ) )
```

Compilation of queries

The query languages of SIQ and LINQ are more expressive than SQL and existing database query libraries for Scala, Scala Query [Sca] and Squeryl [Squ]. We allow nested results, which requires an encoding of complex queries in lower level SQL queries. SIQ's translation involves the Ferry mapping technique [Sch08], which encodes queries with nested results more efficiently than Microsoft's LINQ-to-SQL[3] provider. Chapter 3 precisely defines the SIQ query language, formally describes the required translation steps to SQL, execution and decoding of the relational results.

[1] http://xkcd.com/327/
[2] http://msdn.microsoft.com/en-us/netframework/aa904594
[3] http://msdn.microsoft.com/en-us/library/bb425822.aspx

Compile time guarantees and code lifting

LINQ and SIQ both perform compile time checks on queries, but SIQ provides stronger guarantees. LINQ bases the checks on its unified set of query operators [SQO06] regardless of the particular backend. If an operator is not supported by the chosen backend its use is not caught at compile time, but fails at run time [RF10]. SIQ custom tailors its type safety model to the back end. The implementation is based on the Lightweight Modular Staging (LMS) framework [RO10] which allows a high degree of modular code re-use when adapting type safety to different backends. It also features a basic structure and small core library for run time code lifting, which means providing an abstract syntax tree or expression tree of certain parts of code for further processing at run time.

Chapter 4 describes a prototype implementation of SIQ as a pure library, which performs code lifting without special support by the Scala compiler. We look into how type safety is achieved, how the relevant parts of the LMS framework work and how the compilation into SQL is implemented.

Terminology

This thesis uses a couple of terms equivalently or ambiguously. Depending on the context Scala Integrated Query (SIQ), can mean the general approach, the language in which queries are expressed or the prototype implementation. SIQ query or SIQ expression or SIQ program all mean the same, a query expressed using the query language of SIQ. A record class is a class whose instances are used to store exactly one row of a particular table.

2. Scala and Scala Virtualized

This chapter provides short introductions to Scala and Scala Virtualized.

Scala

Scala is a modern, general purpose programming language, which provides a unique combination of features. It runs on the Java Virtual Machine and is fully interoperable with Java. On the one hand it elegantly fuses object-oriented and functional programming. On the other hand it is statically typed, offers an expressive type system and provides strong compile time guarantees, while at the same time offering a lightweight syntax similar to that of scripting languages like Python. Scala uses type inference within method bodies but explicit typing for method signatures.

The appealing combination of object-orientation and functional programming inspired not only Scala but also other projects like Microsoft F# and GOS [Rum95]. The latter divides programs between a purely functional and a stateful, object-oriented layer. In contrast Scala allows an arbitrary mix of functional and object-oriented constructs within a program and does not currently provide purity guarantees[1].

Type annotations, type inference, mutablity, Predef

Type annotations follow the variable name or method parameter list, separated by a colon. Scala distinguishes mutable variables and immutable values, reflected in the syntax with the keywords val and var. In the following example, foo is mutable and bar is immutable. Mutable variables can be re-assigned to at any time, while immutable values can only be assigned to during definiton. An undefined immutable value is an abstract member of the surrounding class or trait.

```
var foo : Int
val bar = "String"
println( bar )
```

The type of foo is stated explicitly, while the type of bar is inferred from the type of the assigned value. Some methods like println are always imported by default from package Predef.

Classes, singleton objects, traits and case classes

The constructor code of a class is written directly in the body of the class and the constructor parameters are specified as a parameter list of the class. If a parameter is prefixed with

[1] An effect system is under development by Lukas Rytz. https://wiki.scala-lang.org/download/attachments/1310722/effects.pdf

val or var, they automatically become members of the class, public by default but a private modifier can be specified. Scala has native support for singleton objects. In Scala files can contain several classes, traits or objects.

```
class Foo( bar1 : Int, val bar2 : String, private var bar3 : Double ){
      val barsqrt = bar1 * bar1
}
val f = new Foo( 1,2,3 )
object g{
      val bar4 = 6
}
println( f.bar2 + f.barsqrt + g.bar4 )
```

A trait is like a class but does not have a constructor and can be used in multiple-inheritance.

```
trait Foo1{
      val bar1 = 1
}
trait Foo2{
      val bar2 = 2
}
abstract class FooAbstract extends Foo1 with Foo2
trait Foo3
class Foo extends FooAbstract with Foo3
val f = new Foo
println( f.bar2 + f.bar1 )
```

Abstract classes and traits cannot be instantiated. Case classes are Scala's implementation of algebraic data types. They are ordinary classes, but can be used in pattern matching and the Scala compiler automatically generates a couple of methods. Constructor parameters of case classes always become class members. When instantiating a case class the new keyword can be omitted.

```
case class Foo( bar1 : Int, bar2 : Int )
val f = Foo( 1,2 )
println( f.bar1 + f.bar2 )
```

More...

The prototype implementation of SIQ relies on many advanced features of Scala, like implicit conversions and parameters, higher-order methods, covariance and abstract types. [Ode11][2] explains most of the features needed to understand the technical details of the prototype implementation.

Scala Virtualized

The prototype implementation described in Chapter 4 requires Scala Virtualized [CDM+10], an experimental branch of the Scala compiler, which adds certain language features to ease the implementation of Domain Specific Languages (DSL) and embedding of languages into Scala.

[2] http://www.scala-lang.org/docu/files/ScalaByExample.pdf

The SIQ prototype uses two of the new features, infix methods and overloading of ==. Infix methods are similar to extension methods in C#. They allow extending the interface of a class without changing its definition itself. When an infix method overloads an existing method, it takes precedence over it for the argument types it covers. The following example shows how a method can be added to the predefined type Int and how an infix method has higher precedence than the predefined one.

```
def infix_addtwice( i:Int, j:Int ) = i + j + j
assert( 5.addtwice(1) == 7 )
def infix_+( i:Int, j:Int ) = 1
assert( (1 + 1) == 1 )
```

Method == can also be overloaded using an infix method, only that it has a magic name.

```
def __equal( i:Int, j:Int ) = true
assert( 1 == 2 )
```

3. Theory - language definition, translation steps, execution, results

3.1. Overview

This chapter defines the SIQ query language, formally describes the translation steps to SQL, execution strategies and decoding of the relational results. Figure 3.1 shows an overview.

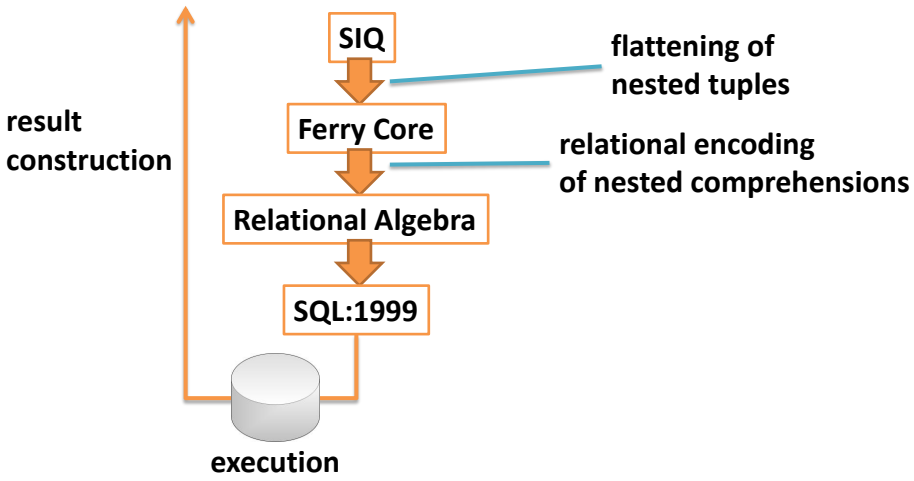


Figure 3.1.: Translation steps overview

The SIQ query language is a subset of Scala and its standard library extended by a small library for additional functionality and an auto-generated library corresponding to a relational database schema. Sections 3.2 and 3.3 define the SIQ query language and how the database schema is integrated.

It is important to know why we chose exactly the shown translation steps to SQL. The choice is motivated by SIQ's data model, which is more expressive than that of SQL and closer to Scala's data model. We support nested collections in the results. The result type of the query shown in chapter 1 is Iterable[Iterable[Employee]]. This particular query groups employees by their work group. SQL however only supports flat results in the form of flat multi-sets of flat records. Nesting, such as a grouping, can not be expressed as such in SQL, which features a GROUP BY operator but requires grouped values to be aggregated (using COUNT, SUM, etc.) to form the same flat result structure. ScalaQuery [Sca] and Squeryl

[Squ] are similar to SQL in that query results are flat collections. Like SIQ, LINQ supports nested collections in the results, but with LINQ-to-SQL such queries can lead to a large number of SQL queries being executed, one for each row of one of the involved tables. Ferry [Sch08] describes how nested comprehensions written in a language called Ferry Core can be encoded more efficiently in Relational Algebra, which can be easily translated to SQL. To re-use that translation recipe, we first translate SIQ queries into Ferry Core, then apply the Ferry mapping and finally translate the resulting Relational Algebra expressions into SQL. Sections 3.4, 3.5 and 3.6 describe these steps formally. Ferry Core only supports flat tuples, but SIQ supports nested tuples and as a consequence we flatten tuples during the translation. We also extend the Ferry mapping by a rule for ordering results.

After the compilation to SQL, queries need to be executed in a database management system. Ferry translates queries with nested results to a tree of Relational Algebra expressions corresponding to the nesting structure of the result type. Possibly several SQL queries have to be executed. They can be submitted to the database one-by-one, as a bundle or lazily. Section 3.7 describes the formal steps for the one-by-one strategy and discusses the alternatives. Finally, section 3.8 explains how the relational results can be decoded and the expected Scala data structure can be constructed.

In this chapter we stay on an abstract level and try to impose only few restrictions on an implementation of the formal description. One such restriction is, that the translation has to happen at run time because of the inlining of values explained in section 3.4, which may come from run time user input.

3.2. The SIQ query language

The query language of SIQ is a purely functional subset of Scala and its standard library with the addition of two small libraries. The only available types are Int, String, Boolean, tuples of all arities and Iterable[T] for T of an available type. String is considered an atomic type in SIQ. Literal values for any of these types are allowed, the factory methods of the List object can be used to create literal values for Iterables. SIQ could be easily extended to support more types like Double but in this work, the types are restricted as stated.

SIQ queries are valid Scala code the Scala compiler can check for validity. Actually the further translation relies on the Scala compiler having type-checked queries.

Queries are embedded into Scala programs. Only those expressions are considered part of the SIQ query, which rely on database data to be evaluated, for example iteration over a database table or comparison of a value with the contents of a database column. Everything else can be evaluated without the database and is considered part of the surrounding Scala program. SIQ programs can use variables of the surrounding Scala program. They are considered free variables in the SIQ program and during the translation the values they are bound to in the surrounding Scala programs are inlined as literals replacing the free variable.

In SIQ type definitions are not possible. They do not depend on the database schema and are automatically considered part of the surrounding Scala program. Variable and method definitions are not allowed in general, but anonymous functions are. The prototype implementation presented later does however support variable and method definition and use, because it inlines uses and calls before the translation. Under this assumption, they can be

considered allowed, but are not handled explicitly by the translation to SQL.

In SIQ, only a few predefined methods on the available types are supported, which we define using the following interface traits. The traits are only meant as a description of the available methods, they are not part of the later presented prototype implementation.

```
trait Iterable[A]{
      /** Concatenates two iterables. */
      def ++ [A] (that: Iterable[A]): Iterable[A]

      /** Applies f to all elements and flattens the result. */
      def flatMap [B] (f: (A) => Iterable[B]): Iterable[B]

      /** Concatenates the elements of an iterable of iterables. */
      def flatten [A] : Iterable[A]

      /** Applies f to all elements and returns result. */
      def map [B] (f: (A) => B): Iterable[B]

      /** Finds the largest element. */
      def max : A

      /** Finds the smallest element. */
      def min : A

      /** The size of this iterable collection. */
      def size : Int

      /** Sums up the elements of this collection. */
      def sum : Int

      /** Removes all elements that do not fulfill a prediacte */
      def withFilter (p: (A) => Boolean): Iterable[A]
}

trait Int{
      def + : Int
      def - : Int
      def * : Int
      def / : Int
      def % : Int
      def ==(i:Int) : Boolean
      def !=(i:Int) : Boolean
      def <=(i:Int) : Boolean
      def >=(i:Int) : Boolean
      def < (i:Int) : Boolean
      def > (i:Int) : Boolean
}

trait String{
      def ==(s:String) : Boolean
      def !=(s:String) : Boolean
      def <=(s:String) : Boolean
      def >=(s:String) : Boolean
      def < (s:String) : Boolean
      def > (s:String) : Boolean
}

trait Boolean{
      def ==(b:Boolean) : Boolean
      def &&(b:Boolean) : Boolean
      def ||(b:Boolean) : Boolean
}

trait Tuple1 [T1]{
```

```
      def _1 : T1
}

trait Tuple2 [T1,T2]{
      def _1 : T1
      def _2 : T2
}

trait Tuple3 [T1,T2,T3]{
      // ... and so on for all tuple arities ...
```

The subset includes several methods from the Scala collections API, which originally use builders to return the most specific collection type[1]. As SIQ only supports Iterable, builders are removed from the signatures [OM09].

List comprehension syntax

By supporting the methods map, flatMap and withFilter, SIQ also supports Scala's for-comprehensions syntax, which is simply desugared to map, flatMap and withFilter calls as explained for example in [Ode11]. The query in chapter 1 could be written semantically equivalent as

```
for( w <- workgroup ) yield ( for(e <- employee; if e.workgroup_id == w.id) yield e )
```

Literals

SIQ supports literals for Int, String, Boolean and tuples. The factory methods of the List object can be used for literal Iterables.

```
List( v1, v2, ... )
```

Extension of the Scala subset

In addition to the Scala subset so far, we add a small library which adds a method orderBy to Iterable. The trait below is meant as an extension of the interface seen before, not a replacement. Just like in SQL, the method allows sorting a collection by a given list of expressions in precedence of appearance accompanied by sorting direction specifiers.

```
trait Iterable[A]{
      def orderBy( o:(A => Order)* ) : Iterable[A]
}
abstract class IOrderable{
  def asc : Order
  def desc : Order
}
implicit def order_enable( r:Int ) : IOrderable
implicit def order_enable( r:String ) : IOrderable
implicit def order_enable( r:Boolean ) : IOrderable
```

The class IOrderable and method order_enable are not considered part of SIQ and may not be used in SIQ programs. They are just implementation details to achieve the desired syntax, which allows queries like

[1] http://www.scala-lang.org/docu/files/collections-api/collections-impl_1.html

```
employee.orderBy( _.age asc, _.name desc )
```

with equivalent semantics to the SQL ORDER BY syntax. The existing ordering methods of Scala collections, sortBy and sortWith are not very well suited for SQL-like ordering by several expressions specifying ascending and descending order. Similar semantics could be achieved by a chain of calls to the sortBy and reverse methods, but we believe such chains to be more complicated to write, to compile to SQL and less intuitive for users familiar with SQL-like ordering.

3.3. Type safe database schema integration

SIQ queries need to be able to refer to database tables and columns. Also we want the Scala compiler to be able to provide compile time guarantees like type safety. In SIQ we achieve this by providing predefined objects and classes that resemble a relational database schema. They can be auto-generated from the database schema in an implementation. During translation to SQL, they are treated specially. During the static analysis of the Scala program, the Scala compiler can use them to type-check queries and to check the existence of the referenced tables and columns. These checks make the important assumption, that this Scala representation of a database schema actually matches the schema of the database the resulting SQL queries are evaluated in. When using SIQ in a software development project it is important to keep this assumption in mind and integrate it into the development process, for example by updating the representation in the build process.

For every table in the database schema we then define what we call a record class, which is a case class capable of storing exactly one row from the particular table. The class has a member for every column of the table with a compatible Scala type. Record classes are very similar to tuples but for convenience allow named access to their members.

We also define a table object that represents the table itself and is exposed to user code as an Iterable parameterized with the record class. Internally it contains metadata about the table like the name and the columns, which is required later during translation.

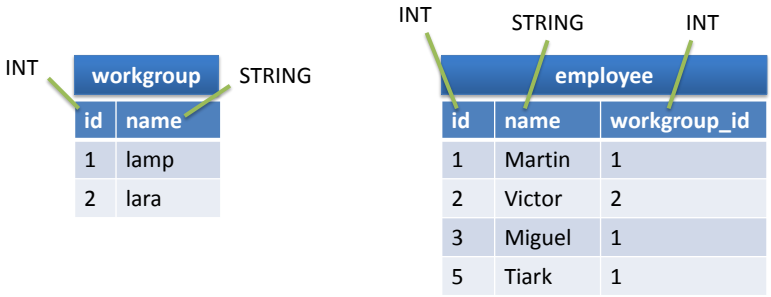


Figure 3.2.: Example tables

Figure 3.2 shows two tables, which we will use as examples throughout this work. For these two tables we would define two record classes Workgroup and Employee and two table objects workgroup and employee, here represented by a vals here.

```
case class Workgroup(
  val id : Int,
  val name : String
)
case class Employee(
  val id : Int,
  val name : String,
  val workgroup_id : Int
)

val workgroup : Iterable[Workgroup]
val employee : Iterable[Employee]
```

Such classes and objects are predefined in programs using SIQ. They extend the available types in SIQ mentioned in the previous section.

The term record class is important to remember, as we use it several times during the coming translation steps to refer to classes like Employee. During the translation there is no conceptual difference between record classes and tuples, only that record classes can only contain atomic-typed values (Int, Boolean, String), while tuples may contain any allowed type including Iterables, tables, tuples and record classes.

3.4. SIQ to Ferry Core

As mentioned before, we first translate SIQ queries into Ferry Core. This chapter defines Ferry Core and formally describes the translation rules from SIQ. The translation relies on the precondition that the Scala program containing the SIQ queries successfully passed the Scala compiler and is thus well formed.

The translation from SIQ to Ferry Core performs two structural changes of the queries. Nested tuple and record class structures are flattened into single flat tuples, because Ferry Core does not support their nesting. When later constructing the result data structure, the nesting structure can be recovered from the expected Scala type.

Expressions that are only valid in the surrounding Scala program, e.g. variables defined outside the SIQ query are inlined to turn the query into a self-contained Ferry Core expression.

For simplicity, the translation assumes, that the source SIQ query is in a canonical form. Methods and operators are expected to be specified using the full dot and parenthesis syntax, e.g. `x.==(y)` rather than `x == y`. Identifier names defined in anonymous functions are expected to be pairwise distinct, so that the translation does not have to deal with overlapping names in nested scopes.

In the translation rules, in SIQ code, v always stands for a Scala identifier. A rule that takes a v as parameter can only be applied to Scala identifiers. Analogously, a stands for an atomic constant, l stands for an Iterable and e stands for any kind of expression including the previously mentioned ones.

3.4.1. Ferry Core

The language Ferry Core has been originally described in [Sch08]. We introduce a slightly modified version, adapted to our use case. Figure 3.3 shows an EBNF grammar of our variant of Ferry Core. Figure 3.4 shows its type system. Compared to the original, we

Ferry Core expressions

e	::=	$\ell \mid v$	atomic literals, variables
	|	$(e,e,\ldots,e) \mid e.n$	tuples, positional access
	|	[e] | []	list literals
	|	table $R\,(c,\ldots,c)$	table access
	|	for v in e	list comprehension with variable binding
		[where a]	optional filter
		[order by $a\ o,\ldots,a\ o$]	optional order
		return e	expression yielded in a comprehension
	|	$e * e \mid f(e,\ldots,e)$	function application

meta variables

e	Ferry Core expression	ℓ	literal
v	variable name	n	tuple position ($\in \{1, 2, \ldots\}$)
c	column name	f	function name ($\in$ {concat,append, length,min,max,sum})
R	table name	$*$	infix operator ($\in \{+, -, <, \ldots\}$)
a	expression with atomic result type	o	an order, asc or desc

Figure 3.3.: FERRY CORE language reference (adapted from [Sch08] and modified)

removed some language constructs not used in our translation and do not distinguish between atomic types. Since we translate a well formed SIQ program to Ferry Core, the translation will result in a well formed Ferry Core program. Hence, we can omit type checking at the level of Ferry Core and do not need to distinguish atomic types. Like the original, we only support list literals with zero or one element. Longer lists are encoded by combining many lists with one element using function append.

t	::=	$(b, b, \ldots, b)$	tuple types
	|	b	nestable types
b	::=	$[t]$	list types
	|	$atomic$	atomic types

Figure 3.4.: FERRY CORE types (adapted from [Sch08] and modified)

Example

Before we formally describe the translation, let's look at an example. The translation source is an SIQ query, for example

```
workgroup.map( w => employee.withFilter( _.workgroup_id == w.id ) )
```

The translation target is a Ferry Core expression. The SIQ query above is translated into the following expression:

```
for w in table workgroup(id, name)
      return ( for e in table employee(id, name, workgroup_id) where w.1 == c.3 return
          (e.1,e.2,e.3) )
```

3.4.2. Auxiliary data types

Among others, the translation rules use lists and map constructs as data structures. We use the following notation.

$[x_1, x_2, ...]$	an ordered list
$\{x_1, x_2, ...\}$	a set
$[k_1 \to x_1, k_2 \to x_2, ...]$	an ordered map with keys $k_1, k_2, ...$

3.4.3. Helper functions

The translation uses a couple of helper functions which we define first.

eval: Inlining expressions

As mentioned before, the translation inlines expressions, which are only valid in the surrounding Scala program, to make queries self-contained. Helper function eval is responsible for this. It evaluates an expression in the context of the surrounding Scala program and returns an SIQ literal. For example: $\mathsf{eval}(\mathsf{Range}(1,4)) = \mathsf{List}(1,2,3)$. Figure 3.5 shows the definition.

$$
\mathsf{eval}(x) = \begin{cases}
... & \\
\text{-1} & \text{if x == -1} \\
0 & \text{if x == 0} \\
1 & \text{if x == 1} \\
2 & \text{if x == 2} \\
... & \\
\text{""} & \text{if x == ""} \\
\text{"a"} & \text{if x == "a"} \\
\text{"b"} & \text{if x == "b"} \\
... & \\
\text{"aa"} & \text{if x == "aa"} \\
... & \\
\mathsf{true} & \text{if x == true} \\
\mathsf{false} & \text{if x == false} \\
\mathsf{List}(\mathsf{eval}(x(0)), \mathsf{eval}(x(1)), ...) & \text{if x:List[\_]} \\
\mathsf{eval}(x.toList) & \text{if x:Iterable[\_]} \\
\mathsf{Tuple1}(\mathsf{eval}(x.\_1)) & \text{if x:Tuple1[\_]} \\
\mathsf{Tuple2}(\mathsf{eval}(x.\_1), \mathsf{eval}(x.\_2)) & \text{if x:Tuple2[\_,\_]} \\
... &
\end{cases}
$$

Figure 3.5.: Helper function eval

is_table, table_name and columns: database schema metadata

Function is_table(t) checks if t is a table object and function table_name(t) returns its name.

In SIQ, database columns are referred to by their name. In Ferry Core tables specify an order of their columns, which are then referred to by their position. We need to translate SIQ's named references to Ferry Cores's positional references. Helper function columns provide the necessary information given a database table object. The result is an ordered map from column names to increasing indices starting with one.

columns(t) = $[c_1 \to 1, c_2 \to 2, ...]$ where c_i are the column names of table object t.

prefix: build up member access paths

prefix is an auxiliary function for flattening tuples. It prefixes every element of a given list with a given value followed by a dot. This allows building a list of member access paths for nested tuples and record classes.

$$\text{prefix}(x, [k_1 \to v_1, k_2 \to v_2, ...]) = [x.k_1, x.k_2, ...] \quad \text{(PREFIX-1)}$$

$$\text{prefix}(x, []) = [x] \quad \text{(PREFIX-2)}$$

3.4.4. Linearizing nested tuples and record classes

$\mathcal{R}$ is an auxiliary function for flattening tuples. It uses previously defined function prefix to build what we call a tuple environment. The tuple environment of a tuple or record class is a map, where the alphabetically sorted keys are all possible member access paths that lead to atomic or list typed values and the map values are a sequence of increasing integers starting with 1. This allows representing the complete tuple nesting structure in the keys and mapping it to a linear structure in the values.

For example (assuming that Customer has the members custkey, name and age):

$$\mathcal{R}[\![(9, ((e : Customer), 3))]\!]^{\Gamma} =$$
$$[\_1 \to 1, \_2.\_1.\text{custkey} \to 2, \_2.\_1.\text{name} \to 3, \_2.\_1.\text{age} \to 4, \_2.\_2 \to 5]$$

The notation $\mathcal{R}[\![e]\!]^{\Gamma} = t$ means that t is the tuple environment of expression e. Γ is a map, where keys are variables defined in the current scope and the values are their tuple environments. This allows checking if a variable name is defined and looking up the tuple environment of the values it is bound to.

The tuple environment of an atomic value is empty. The tuple environment of an Iterable is the same as the tuple environment of its elements. This allows propagating the information up. It is important to keep this in mind when reading rules LIST, FREEVARIABLE, FUNCTIONS-2 and MAP.

$$\mathcal{R}^{\Gamma}[\![a]\!] = [] \quad (\mathcal{R}\text{-ATOMIC})$$

$$\mathcal{R}^{\Gamma}[\![List(e_1, e_2, ...)]\!] = \mathcal{R}^{\Gamma}[\![e_1]\!] \quad (\mathcal{R}\text{-LIST})$$

Rule TUPLE is the core rule of function $\mathcal{R}$. For a possibly nested tuple, it creates the tuple environment using helper function prefix. Operator $+$ means list concatenation here.

$$\frac{[p_1, p_2, ...] = \mathsf{prefix}(\_1, \mathcal{R}^\Gamma[\![e_1]\!]) + \mathsf{prefix}(\_2, \mathcal{R}^\Gamma[\![e_2]\!]) + ...}{\mathcal{R}^\Gamma[\![(e_1, e_2, ...)]\!] = [p_1 \to 1, p_2 \to 2, ...]} \quad (\mathcal{R}\text{-TUPLE})$$

Rules PATHACCESS-1 and PATHACCESS-2 handle tuple and record class member access. If the access path p exists in the tuple environment of expression e, the accessed path points to an atomic or list typed value, which results in an empty tuple environment. If the given access path p does not match exactly but is a prefix of a number of paths in the tuple environment of e, further member access is possible. The suffixes q_1 to q_n are the possible remaining paths.

$$\frac{[..., p \to i, ...] = \mathcal{R}^\Gamma[\![e]\!]}{\mathcal{R}^\Gamma[\![e.p]\!] = []} \quad (\mathcal{R}\text{-PATHACCESS-1})$$

$$\frac{[..., p.q_1 \to i_1, ..., p.q_n \to i_n, ...] = \mathcal{R}^\Gamma[\![e]\!]}{\mathcal{R}^\Gamma[\![e.p]\!] = [q_1 \to i_1, ..., q_n \to i_n]} \quad (\mathcal{R}\text{-PATHACCESS-2})$$

When a variable is accessed, BOUNDVARIABLE looks up the tuple environment in Γ and returns it.

$$\frac{[..., v \to r, ...] = \Gamma}{\mathcal{R}^\Gamma[\![v]\!] = r} \quad (\mathcal{R}\text{-BOUNDVARIABLE})$$

FREEVARIABLE handles free variables in SIQ queries, which are bound in the surrounding Scala program. For table objects, the keys of the tuple environment are the column names, because these are the names of the all atomic members of their elements. For other values, the tuple environment is that of their inlined values.

$$\frac{[v_1 \to r_1, v_2 \to r_2, ...] = \Gamma \qquad v \notin \{v_1, v_2, ...\}}{\mathcal{R}^\Gamma[\![v]\!] = \begin{cases} \mathsf{columns}(v) & \text{if } \mathsf{is\_table}(v) \\ \mathcal{R}^\Gamma[\![\mathsf{eval}(v)]\!] & \text{otherwise} \end{cases}} \quad (\mathcal{R}\text{-FREEVARIABLE})$$

FUNCTION-1 handles functions that return atomic values, which have an empty tuple environment.

$$\frac{f \in \{\mathsf{max}, \mathsf{min}, \mathsf{size}, \mathsf{sum}\}}{\mathcal{R}^\Gamma[\![l.f]\!] = []} \quad (\mathcal{R}\text{-FUNCTIONS-1})$$

$$\frac{f \in \{\mathsf{flatten}\}}{\mathcal{R}^{\Gamma}[\![l.f]\!] = \mathcal{R}^{\Gamma}[\![l]\!]} \quad (\mathcal{R}\text{-FUNCTIONS-2})$$

$$\frac{f \in \{++, \mathsf{withFilter}, \mathsf{orderBy}, +, -, *, /, \%, ==, !=, <=, >=, <, >, \&\&, ||\}}{\mathcal{R}^{\Gamma}[\![e.f(y)]\!] = \mathcal{R}^{\Gamma}[\![e]\!]} \quad (\mathcal{R}\text{-FUNCTIONS-3})$$

For methods map and flatMap, rule MAP defines a new variable v and adds it to the current scope by adding it to Γ.

$$\frac{f \in \{\mathsf{map}, \mathsf{flatMap}\}}{\mathcal{R}^{\Gamma}[\![l.f(v => e)]\!] = \mathcal{R}^{\Gamma+[v \to \mathcal{R}^{\Gamma}[\![a]\!]]}[\![e]\!]} \quad (\mathcal{R}\text{-MAP})$$

3.4.5. Translation rules

This section defines function $\mathcal{N}[\![s]\!]^{\Gamma} = f$ which translates an SIQ query s to Ferry Core expression f. Γ is the same as in $\mathcal{R}$. Function $\mathcal{N}$ uses Γ only to distinguish free and bound variables.

$$\mathcal{N}^{\Gamma}[\![a]\!] = a \quad (\mathcal{N}\text{-ATOMIC})$$

Rules LIST-1 to LIST-3 reflect the fact that Ferry Core only supports list literals with zero or one element. Lists with more elements are encoded using append calls.

$$\mathcal{N}^{\Gamma}[\![\mathsf{List}()]\!] = [] \quad (\mathcal{N}\text{-LIST-1})$$

$$\mathcal{N}^{\Gamma}[\![\mathsf{List}(e)]\!] = [\mathcal{N}^{\Gamma}[\![e]\!]] \quad (\mathcal{N}\text{-LIST-2})$$

$$\mathcal{N}^{\Gamma}[\![\mathsf{List}(e_1, ..., e_n)]\!] = \mathsf{append}([\mathcal{N}^{\Gamma}[\![e_1]\!]], ...\mathsf{append}([\mathcal{N}^{\Gamma}[\![e_n]\!]], [])...) \quad (\mathcal{N}\text{-LIST-3})$$

$$\mathcal{N}^{\Gamma}[\![(e_1, ..., e_n)]\!] = (\mathcal{N}^{\Gamma}[\![e_1]\!], ..., \mathcal{N}^{\Gamma}[\![e_n]\!]) \quad (\mathcal{N}\text{-TUPLE})$$

Rules PATHACCESS-1 and PATHACCESS-2 translate Scala tuple element access within a structure of nested tuples and record classes to Ferry Core tuple access using the tuple environment generated by $\mathcal{R}$. In PATHACCESS-2, p is the prefix of one or several paths in the tuple environment, which means $e.p$ is a tuple or record class. In this case the element access is translated to a flat Ferry Core tuple.

$$\frac{[..., p \to i, ...] = \mathcal{R}^{\Gamma}[\![e]\!]}{\mathcal{N}^{\Gamma}[\![e.p]\!] = e.i} \quad (\mathcal{N}\text{-PATHACCESS-1})$$

$$\frac{[..., p.q_1 \to i_1, ..., p.q_n \to i_n, ...] = \mathcal{R}^{\Gamma}[\![e]\!]}{\mathcal{N}^{\Gamma}[\![e.p]\!] = (e.i_1, ..., e.i_n)} \quad (\mathcal{N}\text{-PATHACCESS-2})$$

Rules FREEVARIABLE-1 and FREEVARIABLE-2 translate table objects to the corresponding Ferry Core representation and inline references to the surrounding Scala program as literals.

$$\frac{[v_1 \to r_1, v_2 \to r_2, ...] = \Gamma \quad v \notin \{v_1, v_2, ...\} \quad \mathsf{is\_table}(v) \quad t = \mathsf{table\_name}(v) \quad [c_1 \to i_1, c_2 \to i_2, ...] = \mathsf{columns}(v)}{\mathcal{N}^\Gamma[\![v]\!] = \mathsf{table}\ \ t\ \ (c_1, c_2, ...)} \quad (\mathcal{N}\text{-FREEVARIABLE-1})$$

$$\frac{[v_1 \to r_1, v_2 \to r_2, ...] = \Gamma \quad v \notin \{v_1, v_2, ...\} \quad \neg\mathsf{is\_table}(v)}{\mathcal{N}^\Gamma[\![v]\!] = \mathcal{N}^\Gamma[\![\mathsf{eval}(v)]\!]} \quad (\mathcal{N}\text{-FREEVARIABLE-2})$$

Ferry Core supports variables defined in comprehensions, just as SIQ. Rule BOUNDVARIABLE just passes through the variable name.

$$\frac{[..., v \to r, ...] = \Gamma}{\mathcal{N}^\Gamma[\![v]\!] = v} \quad (\mathcal{N}\text{-BOUNDVARIABLE})$$

$$\frac{f \in \{+, -, *, /, \%, ==, !=, <=, >=, <, >, \&\&, ||\}}{\mathcal{N}^\Gamma[\![e_1.f(e_2)]\!] = \mathcal{N}^\Gamma[\![e_1]\!]\ f\ \mathcal{N}^\Gamma[\![e_2]\!]} \quad (\mathcal{N}\text{-OPERATOR})$$

Scala methods are translated to their Ferry Core equivalents.

$$\mathcal{N}^\Gamma[\![l.\mathsf{flatten}]\!] = \mathsf{concat}(\mathcal{N}^\Gamma[\![l]\!]) \quad (\mathcal{N}\text{-FLATTEN})$$

$$\mathcal{N}^\Gamma[\![l.\mathsf{size}]\!] = \mathsf{length}(\mathcal{N}^\Gamma[\![l]\!]) \quad (\mathcal{N}\text{-SIZE})$$

$$\mathcal{N}^\Gamma[\![e_1 ++ e_2]\!] = \mathsf{append}(\mathcal{N}^\Gamma[\![e_1]\!], \mathcal{N}^\Gamma[\![e_2]\!]) \quad (\mathcal{N}\text{-APPEND})$$

$$\frac{f \in \{\mathsf{max}, \mathsf{min}, \mathsf{sum}\}}{\mathcal{N}^\Gamma[\![l.f]\!] = f(\mathcal{N}^\Gamma[\![l]\!])} \quad (\mathcal{N}\text{-FUNCTIONS})$$

$$\mathcal{N}^\Gamma[\![l.\mathsf{flatMap}(f)]\!] = \mathcal{N}^\Gamma[\![l.\mathsf{map}(f).\mathsf{flatten}]\!] \quad (\mathcal{N}\text{-FLATMAP})$$

Comprehensions are translated to their equivalents, adding the variable to Γ.

$$\mathcal{N}^\Gamma[\![l.\mathsf{map}(v => e)]\!] = \ \mathsf{for}\ v\ \mathsf{in}\ \mathcal{N}^\Gamma[\![l]\!]\ \mathsf{return}\ \mathcal{N}^{\Gamma+[v\to\mathcal{R}^\Gamma[\![l]\!]]}[\![e]\!] \quad (\mathcal{N}\text{-MAP})$$

$$\mathcal{N}^\Gamma[\![l.\mathsf{withFilter}(v => e)]\!] = \ \mathsf{for}\ v\ \mathsf{in}\ \mathcal{N}^\Gamma[\![l]\!]\ \mathsf{where}\ \mathcal{N}^{\Gamma+[v\to\mathcal{R}^\Gamma[\![l]\!]]}[\![e]\!]\ \mathsf{return}\ v \quad (\mathcal{N}\text{-WITHFILTER})$$

$$\frac{\forall i : o_i \in asc, desc}{\mathcal{N}^\Gamma[\![l.\mathsf{orderBy}(v_1 => e_1\ o_1, v_2 => e_2\ o_2, ...)]\!] = \mathsf{for}\ v\ \mathsf{in}\ \mathcal{N}^\Gamma[\![l]\!]\ \mathsf{order\ by}\ \mathcal{N}^{\Gamma+[v_1\to\mathcal{R}^\Gamma[\![l]\!]]}[\![e_1]\!]\ o_1, \mathcal{N}^{\Gamma+[v_2\to\mathcal{R}^\Gamma[\![l]\!]]}[\![e_2]\!]\ o_2\ \mathsf{return}\ \mathcal{N}^\Gamma[\![v]\!]} \quad (\mathcal{N}\text{-ORDERBY})$$

Rule OTHER is a special rule that only applies, when no other rule applies. This situation implies that o is a Scala construct like a method call that is not a valid SIQ construct, but has to be evaluated in the context of the surrounding Scala program.

$$\mathcal{N}^\Gamma[\![o]\!] = \mathcal{N}^\Gamma[\![\mathsf{eval}(o)]\!] \quad (\mathcal{N}\text{-OTHER})$$

3.5. Ferry Core to Relational Algebra

The next step is the translation from Ferry Core to a Relational Algebra dialect based on [Sch08]. The dialect is shown in figure 3.6. Its specialties are the RowRank and RowNumber operators which allow giving sequence semantics to formerly unordered tables.

This chapter motivates the Ferry translation, describes how it works using an example and then defines the formal translation rules. We used a modified version of Ferry Core so we also need a modified translation. Section 3.5.4 defines the modified translation rules and a new translation rule for ordering. In appendix A we cite the translation rules we use as they are from [Sch08].

Operator	Semantik
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\_,\mathsf{a}_n:\mathsf{b}_n}$	*Projection* to columns b_i renamed to a_i
σ_p	*Selection* of all rows which fulfill predicate p
$\_ \times \_$, $\_ \bowtie_p \_$	*Cartesian Product, Join* on predicate p
$\_ \dot{\cup} \_$, $\_ \setminus \_$	*Disjoint Union, Difference*
δ	*Distinct* removes duplicate rows
$@_{\mathsf{a}:v}$	*Attach* attaches a column a with constant value v
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle}$	*Rowrank* values as a, ordered by $\mathsf{b}_1, \ldots, \mathsf{b}_n$
$\#_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle/\mathsf{c}}$	*Rownumber* values as a, ordered by $\mathsf{b}_1, \ldots, \mathsf{b}_n$, and partitioned by c
$\circledast_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle}$	*Operation application* $*(\mathsf{b}_1, \ldots, \mathsf{b}_n)$, results attached as column a
$\mathsf{agg}_{\mathsf{a}:\langle \mathsf{b}\rangle/\mathsf{c}}$	*Aggregation* of column b grouped by c
⛁_{R}	*Table Reference* to persistent table R
a b c – – –	*Literal Tabelle* defines a transient table

Meta Variables

R Table name a,b,c Column name p Predicate v literal value

$*$ Operation ($\in \{+, -, *, <, >, \ldots\}$)

agg Aggregation function ($\in \{\mathsf{sum}, \mathsf{min}, \mathsf{max}, \mathsf{count}\}$)

Figure 3.6.: Operators of the Relational Algebra dialect, adapted from [Sch08]

3.5.1. Motivation

With the Ferry mapping queries with nested results are translated as several SQL queries with the guarantee that the number only depends on the type. In particular, the number of SQL queries is number of occurrences of the word Iterable in the SIQ result type. If the outermost type is a tuple, the translation leads to yet one more SQL query. If the result type is atomic the query is translated to a single SQL query. The result type of the earlier example query was Iterable[Iterable[Customer]]. Without looking at the query, we can see that it is encoded as two SQL queries just from the type.

This is very different to Microsoft's LINQ-to-SQL provider, where queries with nested collections in the result type can lead to the effect, that one SQL query is executed for the

outermost collection, then one for each element in the result and so on for deeper nesting. This makes the number of SQL queries depend on parts of the result. If the size of a database table grows, this can lead to an increase of required SQL queries for a single LINQ query. [GRS10] shows that the Ferry mapping can lead to significantly better performance.

3.5.2. An informal description using an example

To understand how the translation works, we informally follow the translation steps for our example and explain the necessary concepts on the way. The steps are simplified[2] to ease understanding and do not correspond 1-to-1 to the later presented, formal translation rules. Our example had the Scala result type Iterable[Iterable[Employee]] and was encoded in Ferry Core as

```
for w in table workgroup(id, name) return
      ( for e in table employee(id, name, workgroup_id) where w.1 == e.3 return
          (e.1,e.2,e.3) )
```

The Ferry translation encodes this expression in two relational queries. Based on the sample data shown in figure 3.7, the end results of the two relational queries are shown in figure 3.8. Each row in Query A encodes one element of the outer Iterable, each row of Query B encodes one element of one of the inner Iterables. The correspondence between column item1 of Query A and column iter of Query B determines which inner Iterable a row of Query B belongs to. We will now explain, how we translate the Ferry Core expression into Query A and B from figure 3.8.

workgroup	
id	name
1	lamp
2	lara

employee		
id	name	workgroup_id
1	Martin	1
2	Victor	2
3	Miguel	1
5	Tiark	1

Figure 3.7.: Example data

Boxing

First in the translation is a step called boxing, a preliminary step that adds box-annotations to the input Ferry Core expression at the locations, where it needs to be split into several SQL queries. In our example, the outer comprehension is translated into one SQL query (Query A), but it is not possible to encode the return value as a row, because it is again a comprehension, which requires a table. So here we need to split the query, which is denoted by wrapping the return values in function box. The inner, boxed comprehension is then translated as a separate SQL query. The boxed version of our example is the following semantically equivalent Ferry Core expression.

[2] A full, non-simplified, incremental evaluation graph can be found in appendix D

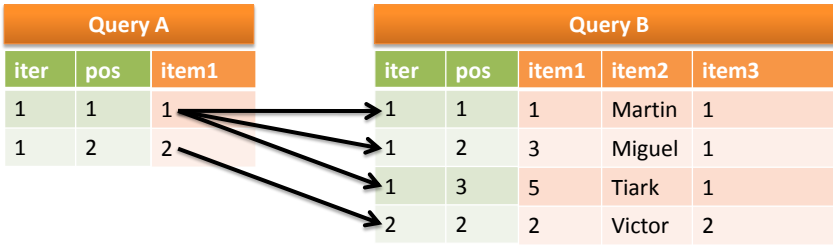


Figure 3.8.: Ferry encoding of the example query

```
for w in table workgroup(id, name) return
      box( for e in table employee(id, name, workgroup_id) where if w.1 == c.3 return
           (e.1,e.2,e.3) )
```

Boxing is implemented using a special type system with two types ROW and TABLE, also called implementation types. List and tables have implementation type TABLE. Tuples and atomic literals have implementation type ROW. Ferry Core's functions and some of the language constructs have fixed return implementation types. For other constructs, Ferry uses type inference to determine the implementation type. The functions and language constructs expect certain implementation types for their arguments. For example the return part of a comprehension expects a ROW typed expression. In our example however, the return part of the outer comprehension contains the inner comprehension, which is a TABLE typed expression. To solve this implementation type mismatch, the translation wraps the inner comprehension in box-call which has argument type TABLE and return type ROW. Boxing is described formally in [Sch08] and also in [Ulr11].

Boxing makes the Ferry translation compositional. In the translation of the outer comprehension the box-call leads to column item1 as seen in figure 3.8. The inner comprehension can then be translated independently provided context information about defined variables and for loop lifting described in the following section.

Loop lifting

After adding box-annotations, we can now start the translation with the outermost expression, a comprehension over table workgroup. Ferry uses a technique called loop lifting to translate comprehensions. Conceptually, loops evaluate parts of code repeatedly under changing variable bindings. In side-effect free languages like SIQ, the order does not matter, the evaluation can be done in parallel. Loop lifting translates comprehensions to Relational Algebra in a way that they can be executed in parallel.

In our example, the body of our outermost comprehension needs to be evaluated once for every element in `workgroup` with different bindings for `w`. The body of the inner comprehension has to be evaluated once for every combination of elements in `workgroup` and `employee`, thus `workgroup.size * employee.size` times.

Loop lifting uses a so called loop context to create as many parallel instances of the expressions in loop bodies as parallel evaluations are required. The loop context is a transient table and the repetition is achieved by building the cartesian product of the current loop

context and the relational encoding of the expression.

Initially, outside any comprehension, the loop context is $\text{loop}_{\text{initial}} =$ iter 1 .

In the body of a comprehension the loop context is changed to reflect the number of parallel evaluations. In our example, `workgroup` has 2 elements and the context in the body our outer comprehension is

$\text{loop}_{\text{outer}} =$ iter 1 2

In the body of the inner comprehension all combinations of employees and work groups have to be considered. `employee` has 4 elements, multiplied with the number workgroups leads to loop context

$\text{loop}_{\text{inner}} =$ iter 1 2 ... 8

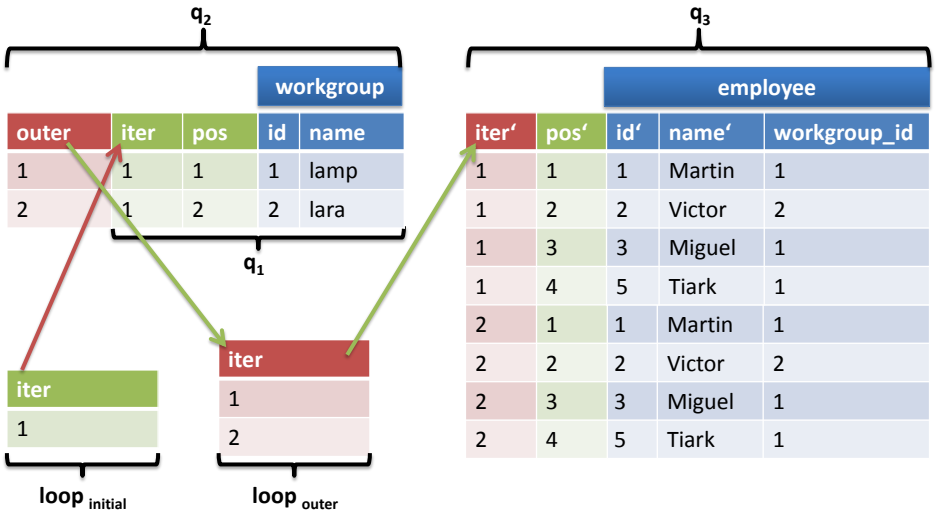


Figure 3.9.: Loop contexts and translation of table references

Figure 3.9 shows how loop contexts are involved in the translation. The reference to table workgroup in the outer comprehension is translated as

$$q_1 = \text{loop}_{\text{initial}} \times \#_{\text{pos:id,name}}\ \text{workgroup}$$

The cartesian product with $\text{loop}_{\text{initial}}$ creates column iter. Column outer is created during translation of the comprehension expression

$$q_2 = \#_{\text{outer:iter,pos}}\ q_1$$

Column outer also becomes the new loop context $\text{loop}_{\text{outer}}$ for translating the comprehension body. box is translated by projecting column outer to item1, which leads to Query A of our results as shown in figure 3.10.

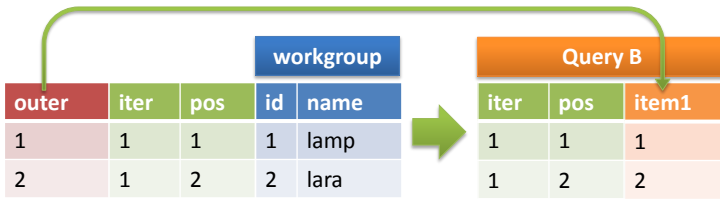


Figure 3.10.: Translation of box and result Query A

The comprehension enclosed in box is then translated as a separate table. Like before, the table reference is translated loop-lifted, taking the current loop context into account as shown in figure 3.9.

$$q_3 = \text{loop}_{\text{outer}} \times \#_{\text{pos}':\text{id}',\text{name}'}\ \text{employee}$$

For translating the inner comprehension, we now create column inner as shown in 3.11.

$$q_4 = \#_{\text{inner}:\text{iter}',\text{pos}'}\ q_3$$

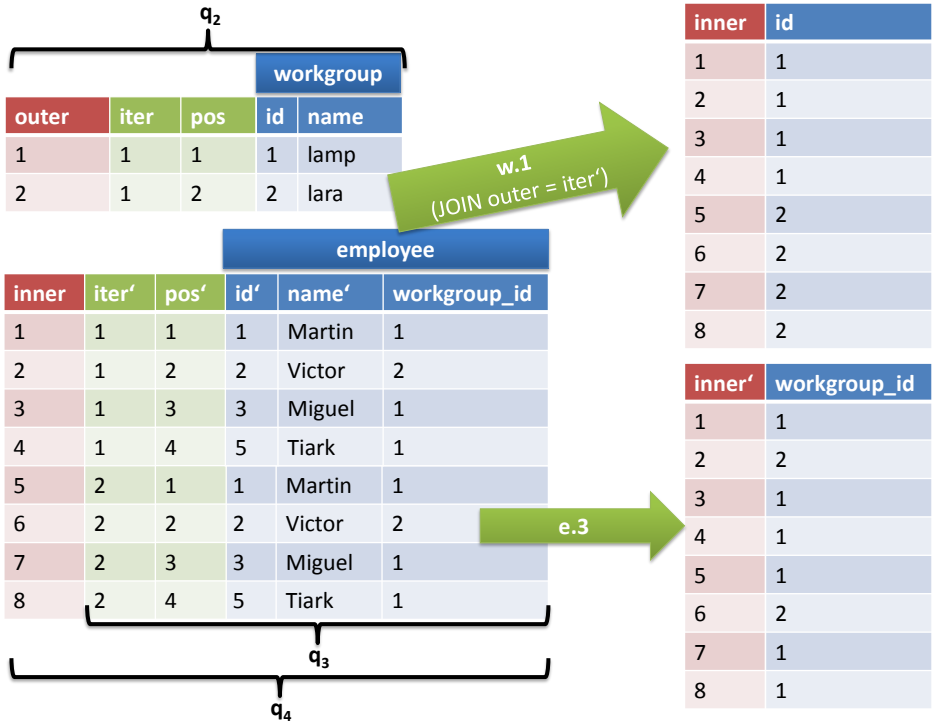


Figure 3.11.: Loop lifting of `w.1` and `e.3`

The `where` part of a comprehension is translated loop lifted like the `return` part. To translate expression `w.1 == e.3`, we first create loop lifted versions of `w.1` and `e.3` as shown in

figure 3.11 and then perform the comparison operation loop lifted as shown in figure 3.12. Which we then filter by boolean column predicate as translation of the `where` construct.

$$q_{\text{where}} = \sigma_{\text{predicate}} \circledcirc_{\text{predicate:id,workgroup\_id}} \left(\left(\pi_{\text{inner,id}} \left(q_4 \bowtie_{\text{outer=iter}'} q_2 \right) \right) \bowtie_{\text{inner=inner}'} \left(\pi_{\text{inner',workgroup\_id}} \, q_4 \right) \right)$$

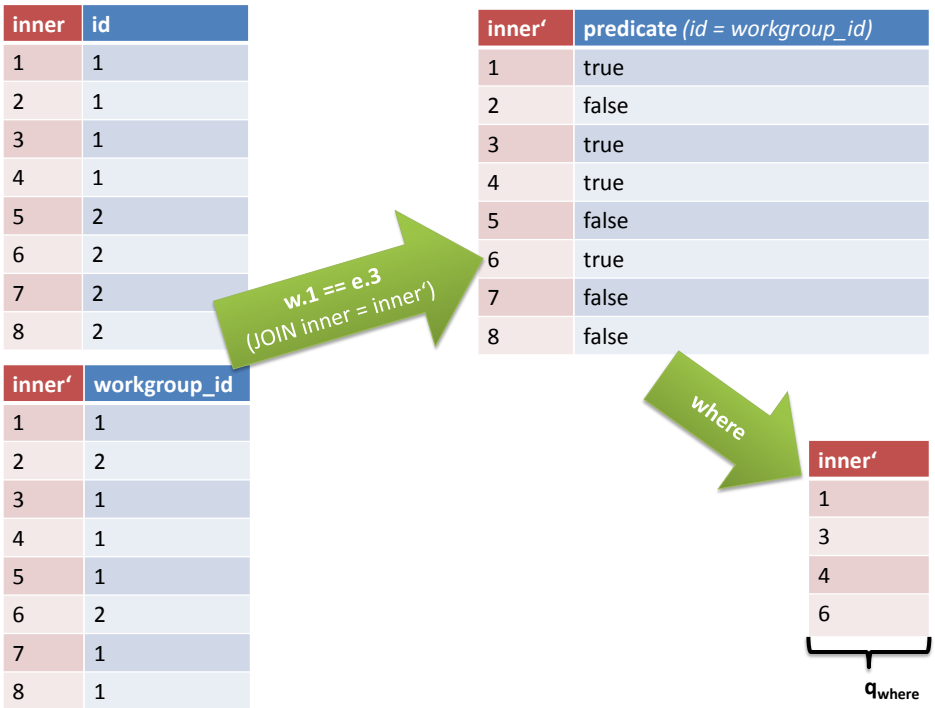


Figure 3.12.: Loop lifting of `w.1 == e.3`

We then translate the `return` part by creating the loop lifted version of `(e.1,e.2,e.3)` as shown in figure 3.13. Joining the loop lifted tuple, q_4 and q_{where} (pair wise) we can use a projection to result in Query B, which concludes the translation of the inner comprehension.

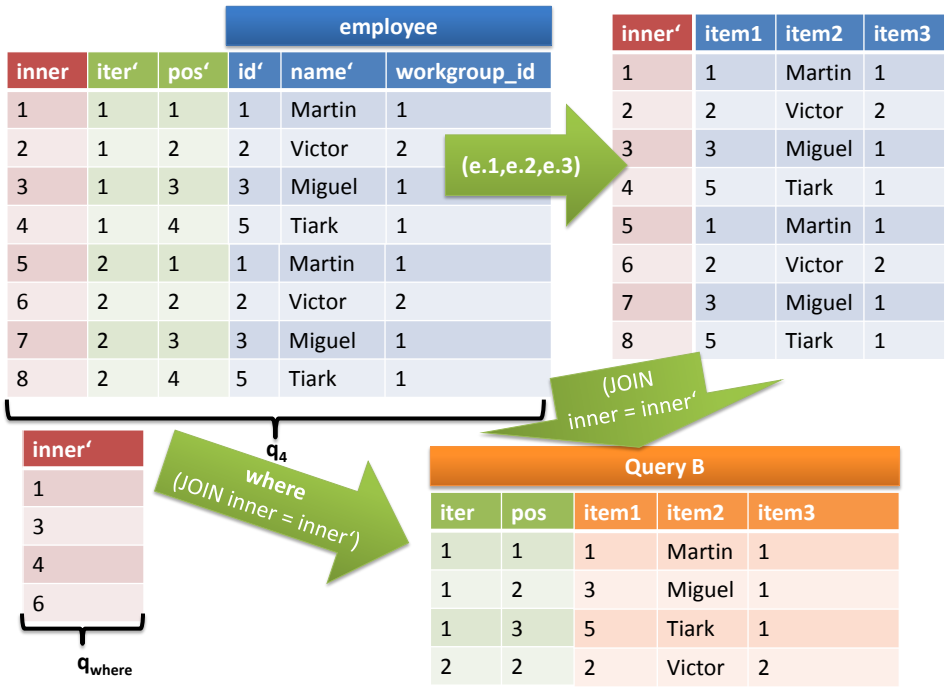


Figure 3.13.: Loop lifting of tuples; filter and Query B

3.5.3. Relational encoding

It is important to understand how the Ferry translation encodes different data types in relational data structures. This understanding is crucial for decoding the relational results later and helpful for understanding the Ferry translation rules.

Lists

List-typed Ferry Core expressions like literal lists and comprehensions are encoded as relational tables. Every row corresponds to one element in the list. The number of columns depends on the element type of the list. Two metadata columns iter and pos are used to encode nesting and list order.

Lists of atomic values Within a list, atomic values are encoded as a single column item1. The following expression returns a list of atomic values. Below the expression we see the relational encoding as a table.

```
for w in table workgroup(id,name) return w.2
```

iter	pos	item1
1	1	"lamp"
1	2	"lara"

Lists of tuples In a list of tuples each tuple is encoded as a single row and every component as a column.

```
for w in table workgroup(id,name) return (w.1,w.2)
```

iter	pos	item1	item2
1	1	1	"lamp"
1	2	2	"lara"

Nesting and comprehensions Nesting of lists is encoded using a table for the outer list and a table for the inner list. The table for the outer list has a column item... with keys that refer to the values of the iter column of the table for the inner list. Every row in the table for the outer list stands for one inner list and the key correspondence to rows of the table for the inner list tells which rows encode the elements of the inner list. An example can be seen in figure 3.8.

Comprehensions are encoded using loop lifting as explained earlier.

Atomic values

A Ferry Core expression which results in a single atomic value is encoded as a table with one row and a column item1 holding the value. This encoding is identical to the encoding of a list with a single atomic value. When later reconstructing the Scala data structure from the relational results we can distinguish these cases by looking at the expected Scala type.

```
length( table employee(id,name,workgroup_id) )
```

iter	pos	item1
1	1	4

Tuples Just like single atomic values a single tuple is encoded as a table with a single row and a column for each value. If the tuple contains lists, some of the columns can contain keys referring to the iter columns of other tables iter column as seen with nested lists.

```
( length(table employee(id,name,workgroup_id)), length(table workgroup(id,name)) )
```

iter	pos	item1	item2
1	1	4	2

3.5.4. Translation rules

We now formally describe the translation from Ferry Core to Relational Algebra. The translation rules are specified using the following notation.

$$\Gamma; loop \vdash e \Rightarrow (q, cols, itbls)$$

e is a Ferry Core expression, $loop$ is a loop context as described earlier. The result, $(q, cols, itbls)$ is called a table information node. Component q is a Relational Algebra expression, which encodes the outermost Ferry Core expression of e. Component $cols$ is a set of all columns of relational expression q excluding metadata columns iter and pos. $itbls$ (inner tables) is an ordered map of column names to other table information nodes. This map encodes the connection between the different relational expressions in the result like the ones seen in figure 3.8. If a column appears as a key in $itbls$ this means that the column holds keys to the iter column of the relational expression contained in the table information node it points to. Γ is a map of bound variable names to the table information nodes they are bound to.

It is important to understand that table information nodes form a tree structure, as the $itbls$ component of one node can point to a number of other nodes.

For the translation we mostly use translation rules from [Sch08]. Descriptions of helper functions as well as the translation rules we use unchanged can be found in appendix A. In the following we define new and modified rules that reflect our small changes and additions to Ferry Core and the translation. We only briefly highlight our changes and additions. Detailed explanations of the translation rules can be found in [Sch08].

Rule TABLEREFEXPR simply reflects the new syntax.

$$\frac{\begin{array}{c} cols = \{\mathsf{cid}(1), \ldots, \mathsf{cid}(n)\} \qquad itbls = [] \\ cmap = [c_1 \mapsto \mathsf{cid}(1), \ldots, c_n \mapsto \mathsf{cid}(n)] \\ q \equiv loop \times \left(\#_{\mathsf{pos}:\langle cmap(c_1),\ldots,cmap(c_n)\rangle}\left(\pi_{\mathsf{cid}(1):c_1,\ldots,\mathsf{cid}(n):c_n}(\circledS_{\mathsf{R}})\right)\right) \end{array}}{\Gamma; loop \vdash \mathsf{table\ R\ (}c_1,\ldots,c_n\mathsf{)}\ (q, cols, itbls)} \quad \text{(TABLEREFEXPR)}$$

Rule MACRO-WHERE maps the extended comprehension syntax to rule IFEXPR.

$$\frac{\Gamma; loop \vdash \mathsf{for}\ v\ \mathsf{in}\ e_1\ OptionalOrderBy\ \mathsf{return\ if}\ p\ \mathsf{then}\ e_2 \Rightarrow (q, cols, itbls)}{\Gamma; loop \vdash \mathsf{for}\ v\ \mathsf{in}\ e_1\ \mathsf{where}\ p\ OptionalOrderBy\ \mathsf{return}\ e_2 \Rightarrow (q, cols, itbls)} \quad \text{(MACRO-WHERE)}$$

IFEXPR is similar to [Sch08], but we removed the else branch of the if expression. This is valid because we only use it to desugar the where part of comprehensions, similar to MACRO-FILTER in [Sch08]. When the condition is met, the expression following then is evaluated and included in the result. If the condition is not met, the element is just skipped and not included in the result.

$$\frac{\begin{array}{l}[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]\,; loop \vdash e_1 \Rightarrow (q_{e_1}, \{c\}, [])\\ loop_{then} \equiv \pi_{iter}\,(\sigma_c\,(q_{e_1}))\\ q_{v_{i_{then}}} \equiv \pi_{\mathsf{iter,pos},cols_{v_i}}\,(q_{v_i} \bowtie_{\mathsf{iter}=\mathsf{iter}'}\,(\pi_{\mathsf{iter}':\mathsf{iter}}\,(loop_{then})))\\ q_{v_{i_{then}}} \vdash itbls_{v_i} \overset{itsel}{\Rightarrow} itbls_{v_{i_{then}}}\\ \Gamma_{then} \equiv \left[\ldots, v_i \mapsto \left(q_{v_{i_{then}}}, cols_{v_i}, itbls_{v_{i_{then}}}\right), \ldots\right]\\ \Gamma_{then}; loop_{then} \vdash e_2 \Rightarrow (q_{e_2}, cols, itbls_{e_2})\\ q \equiv \#_{\mathsf{item}':\langle \mathsf{iter,ord,pos}\rangle}\,(@_{\mathsf{ord}:1}\,(q_{e_2}))\\ q' \equiv \pi_{\mathsf{iter,pos},cols \setminus \mathsf{keys}(itbls_{e_2}),\mathsf{keys}(itbls_{e_2}):\mathsf{item}'}\,(q)\end{array}}{[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]\,; loop \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2 \Rightarrow \left(q', cols, itbls_{e_2}\right)} \quad \text{(IFEXPR)}$$

ORDERBYEXPR is an extension of FOREXPR. It orders the results of a comprehension by a given list of expressions c_1 to c_n with atomic result types. Each expression is evaluated loop lifted in the context of the comprehension and the results are attached as columns order_1 to order_n to the relation which encodes the comprehension results. The result is then ordered by these columns with the provided orders o_1 to o_n to achieve the desired end result.

$$[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]; loop \vdash e_1 \Rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1})$$
$$[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]; loop \vdash c_1 \Rightarrow (q_{c_1}, cols_{c_1}, itbls_{c_1})$$
...
$$[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]; loop \vdash c_n \Rightarrow (q_{c_n}, cols_{c_n}, itbls_{c_n})$$

$$q_v \equiv @_{\mathsf{pos:1}}\left(\pi_{\mathsf{iter:inner},cols_{e_1}}\left(\#_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle} q_{e_1}\right)\right)$$

$$loop_v \equiv \pi_{\mathsf{iter}}(q_v) \qquad \mathsf{map} \equiv \pi_{\mathsf{outer:iter,inner}}\left(\#_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle} q_{e_1}\right)$$

$$\Gamma_v \equiv \left[\ldots, v_i \mapsto \left(\pi_{\mathsf{iter:inner,pos},cols_{v_i}}(q_{v_i} \bowtie_{\mathsf{iter=outer}} \mathsf{map}), cols_{v_i}, itbls_{v_i}\right), \ldots\right]$$

$$\Gamma_v(v \mapsto (q_v, cols_{e_1}, itbls_{e_1})); loop_v \vdash e_2 \Rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2})$$

$$q_{\mathsf{un},0} \equiv q_{e_2} \bowtie_{\mathsf{iter=inner}} \mathsf{map}$$

$$q_{\mathsf{un},1} \equiv \pi_{cols_{e_2}\setminus\{\mathsf{orderiter}\},\mathsf{outer},\mathsf{order\_1}}(q_{\mathsf{un},0} \bowtie_{\mathsf{orderiter=inner}} (\pi_{\mathsf{orderiter:iter,order\_1:item1}}\ q_{c_1}))$$

$$q_{\mathsf{un},2} \equiv \pi_{cols_{e_2}\setminus\{\mathsf{orderiter}\},\mathsf{outer},\mathsf{order\_1},\mathsf{order\_2}}(q_{\mathsf{un},1} \bowtie_{\mathsf{orderiter=inner}} (\pi_{\mathsf{orderiter:iter,order\_2:item1}}\ q_{c_2}))$$

...

$$q_{\mathsf{un},n} \equiv \pi_{cols_{e_2}\setminus\{\mathsf{orderiter}\},\mathsf{outer},\mathsf{order\_1},\ldots,\mathsf{order\_n}}(q_{\mathsf{un},n-1} \bowtie_{\mathsf{orderiter=inner}} (\pi_{\mathsf{orderiter:iter,order\_n:item1}}\ q_{c_n}))$$

$$q_{\mathsf{ordered}} \equiv \#_{\mathsf{pos'}:\mathsf{order\_1}\ o_1,\ldots,\mathsf{order\_n}\ o_n/\mathsf{outer}}\ q_{\mathsf{un},n}$$

$$q'_{e_2} \equiv \pi_{\mathsf{iter:outer,pos:pos'},cols_{e_2}}\left(\pi_{cols_{e_2}\setminus\{\mathsf{order\_1},\ldots,\mathsf{order\_n}\},\mathsf{pos'},\mathsf{outer}}(q_{\mathsf{ordered}})\right)$$

$$\frac{}{[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]; loop \vdash \mathsf{for}\ v\ \mathsf{in}\ e_1\ \mathsf{order\ by}\ c_1\ o_1,\ \ldots,\ c_n\ o_n\ \mathsf{return}\ e_2 \Rightarrow \left(q'_{e_2}, cols_{e_2}, itbls_{e_2}\right)}$$

(ORDERBYEXPR)

3.6. Relational Algebra to SQL

In the next step we need to translate the tree of table information nodes from the previous step into a tree of SQL queries. For this we traverse the tree and translate every contained relational expression to SQL. In the following we first explain how a single relational operator can be translated. Based on this we explain how a complete relational expression possibly consisting of many operators can be translated to a single SQL query. Finally we define function $\mathcal{X}$ which translates a tree of table information nodes to a tree of SQL queries.

The translation recipe described here is very simple. Relational operators are translated individually, when in fact they could be combined and translated together to more compact SQL queries. The current translation suffices as a jigsaw in the process to demonstrate the overall translation from Scala to SQL, but leaves room for future improvement. The pathfinder query optimizer [BGK+05] provides an interesting outlook. It features optimizations on the level of relational algebra and efficient SQL compilation. More about

pathfinder in chapter 5.

3.6.1. Translating relational operators individually

[Wen09a] explains how relational operators can be translated to SQL individually. With a few modifications, we define translation rules based on the descriptions as function $\mathcal{O}[\![r]\!] = s$, which translates the first operator of a relational subexpression e into an SQL query s.

Some of the translation rules have preconditions to be applicable. For example a cartesian product requires the column names of the operand relations to be disjoint. We do not mention the preconditions here, as the translation from Ferry to Relational Algebra results in relational expressions, which satisfy the preconditions. A description of the exact preconditions can be found in [Wen09a]. Our translation rules use a few helper functions.

Function uname(r) provides a unique name for a relational expression r, e.g. t1, which is later used as an alias for the subexpression within the complete SQL query. Function uname gives the same aliases to common subexpressions: $s = r \iff \mathsf{uname(s)} = \mathsf{uname(r)}$. For a table reference, uname returns the name of the table: $\mathsf{uname}(\circledcirc_n) = n$.

Function sch(r) returns a list of column names of relational expression r in alphabetical order. A concrete implementation can infer the columns from the contained relational subexpressions.

Function $\mathcal{P}[\![e]\!]$ translates an expression e that computes a column value, for example an arithmetic expression, to SQL. This translation is trivial and not further explained here.

$$\mathcal{O}[\![\pi_{a_1:b_1,...,a_n:b_n} r]\!] = \text{SELECT } b_1 \text{ AS } a_1, ..., b_n \text{ AS } a_n \text{ FROM uname}(r) \quad \text{(PROJECTION)}$$

$$\mathcal{O}[\![\sigma_p r]\!] = \text{SELECT } * \text{ FROM uname}(r) \text{ WHERE } \mathcal{P}[\![p]\!] \quad \text{(FILTER)}$$

$$\mathcal{O}[\![r \times s]\!] = \text{SELECT } * \text{ FROM uname}(r), \text{uname}(s) \quad \text{(CARTESIANPRODUCT)}$$

$$\mathcal{O}[\![r \bowtie_p s]\!] = \text{SELECT } * \text{ FROM uname}(r), \text{uname}(s) \text{ WHERE } \mathcal{P}[\![p]\!] \quad \text{(JOIN)}$$

$$\mathcal{O}[\![\delta r]\!] = \text{SELECT DISTINCT } * \text{ FROM uname}(r) \quad \text{(DISTINCT)}$$

$$\mathcal{O}[\![\#_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle/\mathsf{c}} r]\!] = \begin{array}{l}\text{SELECT } *, \text{ROW\_NUMBER]() OVER} \\ (\text{ PARTITION BY } c \text{ ORDER BY } b_1, ..., b_n) \text{ AS } a \text{ FROM uname}(r)\end{array} \quad \text{(ROWNUMBER-1)}$$

$$\mathcal{O}[\![\#_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle} r]\!] = \text{SELECT } *, \text{ROW\_NUMBER() OVER (ORDER BY } b_1, ..., b_n) \text{ AS } a \text{ FROM uname}(r) \quad \text{(ROWNUMBER-2)}$$

$$\mathcal{O}[\![\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle} r]\!] = \text{ SELECT } *, \text{DENSE\_RANK() OVER (ORDER BY } b_1, ..., b_n) \text{ AS } a \text{ FROM uname } (r) \quad \text{(ROWRANK)}$$

$$\mathcal{O}[\![@_{a:v} r]\!] = \text{ SELECT } *, v \text{ AS } a \text{ FROM uname}(r) \quad \text{(ATTACH)}$$

$$\mathcal{O}[\![r \dot\cup s]\!] = \text{ SELECT } * \text{ FROM uname}(r) \text{ UNION ALL SELECT } * \text{ FROM uname}(s) \quad \text{(DISJOINTUNION)}$$

$$\mathcal{O}[\![r \setminus s]\!] = \frac{[a_1, ..., a_n] = \text{sch}(r) \quad sch(s) = sch(r)}{\begin{array}{l}\text{SELECT } * \text{ FROM uname}(r) \text{ WHERE NOT EXISTS} \\ (\text{ SELECT } * \text{ FROM uname}(s) \text{ WHERE } r.a_1 = s.a_1 \text{ AND ... AND } r.a_n = s.a_n)\end{array}} \quad \text{(DIFFERENCE)}$$

$$\mathcal{O}[\![\circledast_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle} r]\!] = \text{ SELECT } *, \mathcal{P}[\![\circledast(b_1, ..., b_n)]\!] \text{ AS } a \text{ FROM uname}(r) \quad \text{(OPERATORAPPLICATION)}$$

$$\mathcal{O}[\![\text{agg}_{\mathsf{a}:\langle \mathsf{b}\rangle/\mathsf{c}} r]\!] = \text{ SELECT } agg(b) \text{ AS } a \text{ FROM uname}(r) \text{ GROUP BY } c \quad \text{(AGGREGATION)}$$

$$\mathcal{O}[\!\!\begin{array}{|c|c|c|}\hline \mathsf{c}_1 & \ldots & \mathsf{c}_n \\ \hline v_{1,1} & \ldots & v_{1,n} \\ \ldots & \ldots & \ldots \\ v_{m,1} & \ldots & v_{m,n} \\ \hline \end{array}\!\!]\!] = \text{ VALUES } (v_{1,1}, ..., v_{1,n}), ..., (v_{m,1}, ..., v_{m,n}) \quad \text{(LITERALTABLE)}$$

Rule LITERALTABLE is not valid SQL:1999 but valid SQL-2003, as there is no such construct in SQL:1999. It is compatible with the PostgreSQL DBMS.

3.6.2. Translating a complete relational expression

Now that we have translation rules for relational operators, we can combine them to translate whole expressions. For this we first define helper function $\text{topsort}(r)$, which returns a list of all subexpressions within r, in a topological order with duplicates removed. For example

$\text{topsort}(\pi_{\text{name}}\ \sigma_{\text{custkey}=1}\ \text{customer}) = [\text{customer}, \sigma_{\text{custkey}=1}\ \text{customer}, \pi_{\text{name}}\ \sigma_{\text{custkey}=1}\ \text{customer}]$.

We now define function $\mathcal{S}[\![r]\!] = s$ which translates a relational expression r to an SQL query s.

$$\mathcal{S}[\![r_n]\!] = \frac{[r_1, ..., r_n] = \text{topsort}(r_n) \quad [c_{1,1}, ..., c_{1,k}] = \text{sch}(r_1) \quad ... \quad [c_{n,1}, ..., c_{n,k}] = \text{sch}(r_n)}{\begin{array}{l}\text{WITH} \\ \text{uname}(r_1)(c_{1,1}, ..., c_{1,k}) \text{ AS } (\mathcal{O}[\![r_1]\!]), \\ ... \\ \text{uname}(r_n)(c_{n,1}, ..., c_{n,k}) \text{ AS } (\mathcal{O}[\![r_n]\!]) \\ \text{SELECT } * \text{ FROM uname}(r_n)\end{array}} \quad (\mathcal{S})$$

3.6.3. Example

The expression $\pi_{\mathsf{name}}\ \sigma_{\mathsf{custkey}=1}\ \text{⛁}\,\mathsf{customer}$ containing a projection, a filter and a table reference will be will be compiled into the following SQL query.

```
WITH
      t1 (custkey, name) AS (SELECT custkey, name FROM customer WHERE custkey = 1),
      t2 (name) AS (SELECT name FROM t1)
SELECT * FROM t2
```

3.6.4. Translating the tree of table information nodes

Using function $\mathcal{S}$, we can now define the translation from a tree of table information nodes to a tree of SQL queries. We define the translation as function $\mathcal{X}[\![(q, cols, itbls)]\!] = (sql, nested)$, where sql is an SQL query and $nested$ is a map of column names to related query encoding the nesting as before.

$$\mathcal{X}[\![(q, cols, [])]\!] = (\mathcal{S}[\![q]\!], []) \qquad (\mathcal{X}\text{-1})$$

$$\frac{[c_1 \to i_1, c_2 \to i_2, ...] = itbls}{\mathcal{X}[\![(q, cols, itbls)]\!] = (\mathcal{S}[\![q]\!], [c_1 \to \mathcal{X}[\![i_1]\!], c_2 \to \mathcal{X}[\![i_2]\!], ...])} \qquad (\mathcal{X}\text{-2})$$

3.7. Execution

The last translation step resulted in a tree of SQL:1999 expressions, each of which can be executed in a DBMS of our choice. This leads to a tree of result tables, one for every SQL query. Finally the result data structure matching the expected Scala type needs to be constructed from the tree of tabular results.

The execution strategy chosen here is just one way to do it. We execute every SQL query individually while traversing the tree. There are potentially better solutions. A different strategy would be submitting all queries to the database at once as a bundle, so that the DBMS could potentially optimize execution across queries. Yet another strategy would be executing queries lazily, only when the particular data is actually requested.

3.7.1. Retrieving a single result table

Every SQL query in the tree of SQL queries needs to be executed in the DBMS. For this we define the helper function $\mathsf{exec}(sql) = [[\mathsf{iter} \to i_1, \mathsf{pos} \to p_1, item1 \to v_{1,1}, item2 \to v_{1,2}, ...], ..., [\mathsf{iter} \to i_n, \mathsf{pos} \to p_n, item1 \to v_{n,1}, item2 \to v_{n,2}, ...]]$, which executes a single SQL query sql and returns the result table in the form of a list of maps of column names to values ordered by columns iter and pos or in other terms $i_j <= i_{j+1} \wedge (i_j = i_{j+1} \Rightarrow p_j < p_{j+1})$.

3.7.2. Retrieving the result tree

To transform the tree of SQL expressions to the tree of results it is traversed and every expression executed. We define the function $\mathcal{T}$ to perform this task.

$$\mathcal{T}[(sql, [])] = (\mathsf{exec}(sql), [])$$

$$\mathcal{T}[(sql, [\mathsf{item1} \rightarrow v_1, \mathsf{item2} \rightarrow v_2, ...])] = (\mathsf{exec}(sql), [\mathsf{item1} \rightarrow \mathcal{T}[v_1], \mathsf{item2} \rightarrow \mathcal{T}[v_2], ...])$$

3.8. Constructing the result data structure

In the end, we want to have a data structure that conforms to the type of the SIQ expression we compiled to SQL. The method we chose here is a type driven construction of this result data structure. We look at the expected SIQ type using run time type information and create the data structure accordingly from the tree of tabular results.

For this we define the function $\mathcal{Y}[\![(iter, type, values)]\!] = result$ where $type$ is the expected SIQ type, $values$ is a tree of SQL queries as received from the previous translation step and $iter$ is an index denoting the current list element. We start the recursion with $iter = 1$, $type$ as the SIQ type of the compiled expression, and $values$ as the tree of results.

Function $\mathcal{Y}$ processes tables. $\mathcal{X}$ processes single rows. If the expected type is atomic, the table will have a single row with a column item1 which contains the value. $\mathcal{Y}$-1 covers this case.

$$\frac{type \in \{\mathsf{Int}, \mathsf{String}, \mathsf{Boolean}\} \quad ([[\mathsf{iter} \rightarrow i, \mathsf{pos} \rightarrow p, \mathsf{item1} \rightarrow v_1]], nested) = values}{\mathcal{Y}[\![(i, type, values)]\!] = \mathsf{cast}(type, v_1)} \quad (\mathcal{Y}\text{-1})$$

If the expected type is a tuple. The table will contain a single row. $\mathcal{Y}$-2 passes this row to $\mathcal{Z}$ for further processing, passing 1 as first parameter to tell $\mathcal{Z}$ that the value can be found in the first column following iter and pos.

$$\frac{(index, result) = \mathcal{Z}[\![(1, type, row, nested)]\!]}{\mathcal{Y}[\![(iter, (t_1, ..., t_n), ([row], nested))]\!] = result} \quad (\mathcal{Y}\text{-2})$$

In case an Iterable is expected, the table will contain one row for each element. Due to the Ferry encoding, the table can contain more rows, which represent elements from other lists. So $\mathcal{Y}$-3 must only look at those rows, there the iter column corresponds to the element indicator i. It passes each of these rows to $\mathcal{Z}$ for further processing.

The function $\mathcal{Z}$ processes single rows, or single elements within a row. The first parameter tells the position of the element to be processed, type tells the expected type, row is the row to be processed and nested are the tables corresponding to contained lists if any.

$$\frac{row_1 = [\mathsf{iter} \to i, ...] \; ... \; row_n = [\mathsf{iter} \to i, ...] \quad (i_1, r_1) = \mathcal{Z}[\![(1, t, row_1, nested)]\!] \; ... \; (i_n, r_n) = \mathcal{Z}[\![(1, t, row_n, nested)]\!])}{\mathcal{Y}[\![(i, Iterable[t], ([..., row_1, ..., row_n, ...], nested))]\!] = List(r_1, ..., r_n)} \quad (\mathcal{Y}\text{-3})$$

$\mathcal{Z}$-1 covers the case that an atomic value is expected. I uses helper function cast to cast the value in the column at position start to the expected type.

$$\frac{type \in \{\mathsf{Int}, \mathsf{String}, \mathsf{Boolean}\} \quad [iter \to i, pos \to p, c_1 \to v_1, c_2 \to v_2, ...] = row}{\mathcal{Z}[\![(start, type, row, nested)]\!] = (start + 1, \mathsf{cast}(type, v_{start}))} \quad (\mathcal{Z}\text{-1})$$

$\mathcal{Z}$-2 handles the case that an iterable is expected, which means a sublist of the list processed by the surrounding $\mathcal{Y}$ call. $\mathcal{Z}$-2 looks up the item indicator v_{start} and the table storing the data of the sublist and passes them to $\mathcal{Y}$ for processing of the whole table.

$$\frac{[iter \to i_1, pos \to p_1, c_1 \to v_1, c_2 \to v_2, ...] = row \quad [..., c_{start} \to table, ...] = nested}{\mathcal{Z}[\![(start, Iterable[T], row, nested)]\!] = (start + 1, \mathcal{Y}[\![(v_{start}, Iterable[T], table)]\!])} \quad (\mathcal{Z}\text{-2})$$

$\mathcal{Z} - 3$ handles the case of an expected tuple. A single row can encode a whole structure of nested tuples. Each column corresponds to a single non-tuple value contained in the nested structure. $\mathcal{Z}-3$ builds up the expected tuple structure recursively and consumes the values of the row one by one, always increasing a counter when an element is processed.

$$\frac{(s_1, r_1) = \mathcal{Z}[\![(s_0, t_1, row, nested)]\!] \quad ... \quad (s_n, r_n) = \mathcal{Z}[\![(s_{n-1}, t_n, row, nested)]\!]}{\mathcal{Z}[\![(s_0, (t_1, ..., t_n), row, nested)]\!] = (s_n, (r_1, ..., r_n))} \quad (\mathcal{Z}\text{-3})$$

If the expected type is a record class, a variant of rule $\mathcal{Z} - \ni$ is used. Then $t\_1, ..., t\_n$ correspond to the types of the members of the record class and an instance of the class is constructed for the return value instead of a tuple. The values $r\_1, ..., r\_n$ are passed as arguments to the constructor.

4. Implementation technique and prototype

This chapter describes our implementation technique, its advantages and disadvantages. We first give an overview and describe the general idea. We then look at the implementation from a users point of view and finally show implementation internals.

A core goal of our implementation is to provide strong compile time guarantees. Microsoft LINQ uses a unified query language, which defines a set of query operators [SQO06]. It uses compiler support to lift queries to expressions trees. Backend providers supplied as run time libraries map these trees to queries in their particular backend. But not every provider supports every operator, including LINQ-to-SQL[1] and LINQ-to-Entities[2]. If an unsupported operator is used, a run time exception is thrown [RF10].

Using the modularity provided by the Lightweight Modular Staging (LMS) framework [RO10] allows us to do better. We can define modules for different components of our query language and use suitable compositions to implement customized type safety for different backends. This chapter describes the most important modules of SIQ and how type safety is achieved reflecting the specifics of SQL:1999. In short, we map the constraints into the Scala type system. We also give an example how type safety can be adjusted to reflect specifics of MySQL's SQL dialect.

Customizing the query language like this would be complicated in a scenario like LINQ, where the compiler would need to be changed. Making use of certain Scala language features, LMS provides a basis for implementing this modularity as a pure library, without changing the Scala compiler. This allows query language customization for library developers with the additional advantages that a library

- does not add complexity to the compiler
- can be understood without any knowledge about the Scala compiler's internals
- is easier to distribute than a compiler plugin

In the prototype, we use LMS solely for code lifting and modular type safety. The implementation of the compilation and execution steps described in chapter 3 starts with LMS based expression trees as initial input, but is otherwise independent of LMS. This means we only use LMS only as a front end component of the prototype, which could be replaced by a different front end, for example one with a ScalaQuery [Sca] compatible API.

4.1. A users point of view

We now show what using the SIQ prototype looks like and explain the necessary concepts on the way. After the required imports one can write a query like the following. (The

[1]Standard Query Operator Translation: http://msdn.microsoft.com/en-us/library/bb399342.aspx
[2]Supported and Unsupported LINQ Methods: http://msdn.microsoft.com/en-us/library/bb738550.aspx

database connection details are currently hard coded in the prototype, which of course needs to be changed in the future).

```
import siq.dsl.dsl._
import siq.dsl.dsl.tables.employee
import siq.dsl.dsl.implicits._

val employee_names_query = for( c <- employee ) yield c.name // :Rep[Iterable[String]]
```

The value `employee_names_query` does not yet contain the results of executing the query but a value of type Rep[Iterable[String]], representing the query itself. Being a user of SIQ, we do not need to care about the internals of this value. The only thing we need to know is that we can execute it and retrieve the results using method fromdb.

```
val employee_names = employee_names_query.fromdb // :Iterable[String]
```

Alternatively we could use `employee_names_query` as a subquery and as input to other queries without executing it.

4.1.1. Embedding SIQ programs into Scala

SIQ queries are embedded into larger Scala programs. It is important to understand how we can distinguish the Scala program, which runs on the JVM, from the SIQ program, which is compiled to SQL. Based on LMS, we use a type-based distinction. While the Scala code uses the ordinary Scala types like Int and String, in SIQ queries we use Rep[Int] and Rep[String]. Rep[T] is a generic type, which we use as a wrapper for the Scala types T allowed in SIQ (as defined in 3.2). From now on we refer to these types as **Rep-types**. It is important to know that Rep-types are just interface types hiding the actual objects behind them.

Our implementation newly defines all the methods available in SIQ based on Rep-types in parallel to their original definitions defined on ordinary Scala types. Let's look at an example. The SIQ query language described in 3.2 includes

```
trait Boolean{
      def &&(b:Boolean) : Boolean
      def ||(b:Boolean) : Boolean
}
```

the implementation however conforms to this interface

```
trait Rep[T <: Boolean]{
  def &&( r:Rep[T] ) : Rep[T]
  def ||( r:Rep[T] ) : Rep[T]
}
```

The interface is only meant as a description. It is not part of the actual implementation, but a user can use methods `&&` and `||` as if it was.

Newly defining the SIQ methods on Rep-types in parallel to their Scala equivalents allows users to think about SIQ just as if it was Scala because names, structure and syntax are the same and Scala's type inferencer relieves a user from writing Rep-type type annotations in most cases. At the same time Rep-types give us several opportunities. First, we can distinguish Scala from SIQ expressions by their type. Something that is of a Rep-type resembles a query and can be executed in the database. Second, we can exactly control

the methods supported by SIQ by implementing only these on Rep-types. The Scala type-checker will make sure that only supported methods are used on SIQ expressions. Third, providing new definitions for all methods in SIQ allows us to implement them the way we need to in order to return expression trees hidden behind a Rep-type.

It is interesting to understand the meaning of method fromdb in the context of Rep-types. fromdb turns a value of type Rep[T] into a value of type T. In other words it moves something from the database world of Rep-types into Scala world of ordinary types, by compiling it to SQL, executing it and creating the result data structure behind the scenes. As we have seen, `employee_names_query` has type Rep[Iterable[String]] and invoking its fromdb methods turns it into an Iterable[String].

4.1.2. Integrating the database schema

Section 3.3 explained how we can refer to database tables and columns in SIQ queries. Using the SIQ prototype, we first need to import the table objects into the local scope.

```
import siq.dsl.dsl.tables.employee
```

Queries can then be expressed over tables as seen before. Like every type in SIQ, tables use Rep-types. Object `employee` is of type Rep[Iterable[Employee]], the element type within the query is Rep[Employee] and its members have atomic Rep-types like Rep[String].

Appendix B shows the folder structure of the prototype. File Schema.scala contains the code for all table objects and record classes. It is auto-generated from metadata about the database schema. While it is a future goal to obtain the metadata by reflecting the database schema, it is currently hard coded in the source file of the code generator, generator.scala in the following format.

```
ListMap(
  "employee" -> Meta(
    table = "employee",
    fields = ListMap(
      "id" -> "Int",
      "name" -> "String",
      "workgroup_id" -> "Int"
    ),
    keys = List("id")
  ),
  "workgroup" -> Meta(
    table = "workgroup",
    fields = ListMap(
      "id" -> "Int",
      "name" -> "String"
    ),
    keys = List("id")
  ),
  ...
)
```

4.1.3. Integrating Scala values

We have seen how fromdb can be used to turn a Rep-type into an ordinary Scala type (by executing it). We also need to be able to turn values of ordinary Scala types into Rep-types in order to be able to use them in queries. These values could come from a computation,

from user input or from a literal. For example we might want to find all employees with ids smaller than 4.

```
for( e <- employee; if e.id < 4 ) yield e
```

4 is an ordinary Scala type, an Int. `e.id` is of type Rep[Int]. As mentioned before we define all methods on Rep-types and accordingly, method `<` expects a Rep-typed argument for the right hand side.

Our prototype comes with implicit conversions, which convert supported Scala types to Rep-types to do these transformations. They can be imported as seen before using

```
import siq.dsl.dsl.implicits._
```

One way to think about these conversions is a transfer of the value from the JVM heap into the DBMS. In practice this does not happen at the moment of implicit conversion, but later, when compiling the query to SQL.

For converting Iterables we cannot use implicit conversions. The following query does not achieve the desired effect.

```
val employee_queries = for( id <- List(1,3); e <- employee; if e.id == id ) yield e
```

The reason is that it is desugared to a flatMap call on `List`, which is already defined, but does not expect an argument involving Rep-types. Currently Scala's type inferencer is not capable of figuring out that an implicit conversion is required to find a definition of flatMap that is valid here. Instead we need to use an explicit conversion in SIQ.

```
val employee_queries = for( id <- List(1,3).todb; e <- employee; if e.id == id ) yield e
```

Method todb can be used to convert any supported, ordinary Scala typed value into a Rep-typed value explicitly.

4.1.4. Compile time guarantees

For a user it is also interesting to know about the compile time guarantees SIQ provides. They fall into two categories. We guarantee that SIQ programs are compiled into well formed SQL queries. The compilation makes sure that the number of generated SQL queries only depends on the result type, which is known at compile time.

Query wellformedness can be divided into several properties:

Syntactic correctness

The static typing in cooperation with the presented translation recipe from Scala to SQL guarantees that every SIQ program compiles into syntactically correct SQL queries. Incorrect programs are rejected by the Scala compiler.

Correctness with regard to the database schema

Under the assumption that table objects and record classes are in sync with the actual database schema, SIQ guarantees that only valid tables and columns are referenced in queries.

Operation validity and type safety

SIQ explicitly defines its query language. Only those methods included in the language are defined on Rep-types, which makes sure that only these methods are used in queries and only for the defined types.

4.2. Behind the scenes

Now that we have a basic understanding of how to use Scala Integrated Query, it is time to look behind the scenes. We will first look at the relevant parts of the Lightweight Modular Staging framework, how it works, and how SIQ uses it. We will then discuss how we achieve type safety and how different constructs in queries are lifted to expression trees, like comprehensions, tuples and operators. Finally we will see how the compilation to SQL, execution and result construction are implemented.

4.2.1. Lightweight Modular Staging

The Lightweight Modular Staging (LMS) framework [RO10] facilitates the implementation of type safe, embedded DSL's in Scala, especially those that use a variant of Scala for the expression of domain specific programs. It eases type-safe embedding, code-lifting, common subexpression elimination and code-generation.

Modules

For code lifting, LMS specifies a structure for modules to lift Scala methods to expression trees. The framework already includes modules for common operations like arithmetic expressions, which we did not use. Instead we experimented with an approach of generating the code of modules from a concise definition. Some additional LMS modules in SIQ were hand-written.

The separation into modules allows the composition of several slightly different query languages that correspond the the specifics of different backends, as mentioned before. This is particularly helpful in the context of query languages, as many SQL implementation have many small differences. For example, PostgreSQL has strong typing for operator arguments, while MySQL has weak typing. This difference can be reflected by different modules, as we will see in section 4.2.2.

An LMS module consists of two traits, an interface trait and an implementation trait. Let's look at an example.

```
trait IStringOps extends Base {
  def infix_+( l:Rep[String], r:Rep[String] ): Rep[String]
}
trait StringOps extends IStringOps with BaseExp{
  case class Concat(x: Exp[String], y: Exp[String]) extends Def[String]
  def infix_+( l:Exp[String], r:Exp[String] ): Rep[String] = Concat(l, r)
}
```

Here, IStringOps is the interface trait, and StringOps is the implementation trait. Together they form a module for lifting the + method of Strings. The implementation trait defines a case class to represent the operation in the expression tree of the lifted query. The

main object of SIQ inherits the implementation trait, but only exposes the interface to user code.

LMS core: code lifting and common subexpression elimination

To understand the example module in detail, we also need to look at the inherited traits. The following code shows Base, BaseExp and Expressions, the core components of the LMS framework and the only part of the framework SIQ actually uses (with a few modifications to the code explained later). Figure 4.1 visualizes some of the involved relationships.

```
trait Base {
  type Rep[+T]
  implicit def unit[T](x: T): Rep[T]
  ...
}
trait BaseExp extends Base with Expressions {
  type Rep[+T] = Exp[T]
  implicit def unit[T](x: T) = Const(x)
  ...
}
trait Expressions {
      abstract class Exp[+T]
  trait Def[+T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](val id: Int) extends Exp[T]
  implicit def toAtom[T](d: Def[T]): Exp[T] = // ... left out here
  ...
}
```

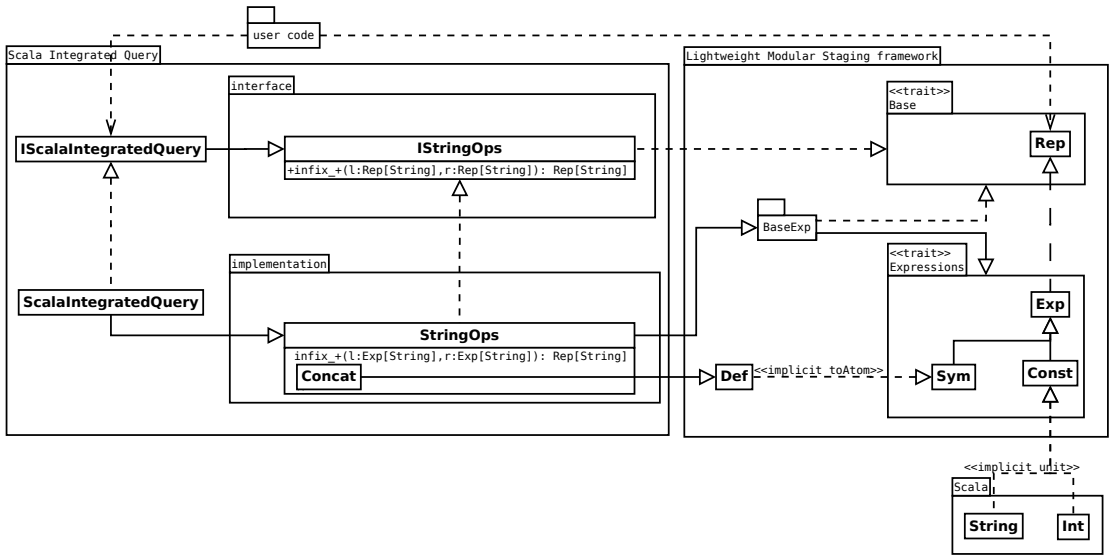


Figure 4.1.: Lightweight Modular Staging framework architecture

We can see that in trait Base, Rep is an abstract type, which is defined in BaseExp as concrete type Exp inherited from trait Expressions. This means that in the interface trait,

which is exposed to the user, nothing particular is known about type Rep, except that it has a type parameter and is covariant in it.

In the implementation trait of our example, the method infix_+ implements the abstract method infix_+ of the interface trait, because Rep is defined as Exp. At first glance, the method implementation looks like it returns an instance of class Concat, but looking closer we can see that the return type is a Rep-type, but Concat is a subtype of Def[String]. To make it work out, implicit conversion toAtom is applied, which puts the Concat instance into a symbol table and returns the corresponding symbol instead, which is an instance of Sym, which is an Exp respectively Rep. If an equivalent object was already in the symbol table, the existing symbol for it is returned. (Remember that case class instances are compared by the values of their members instead of object identity).

Let's put what happens into the larger picture. The creation of the Concat instance is the code lifting of method `+`. The conversion from Def[T] to Rep[T], where the Concat object is put into the symbol table implements common subexpression elimination as equivalent instances are detected and the same symbol is returned. The overall separation between interface trait and implementation trait makes sure that user code only depends on Rep-types, but not on the internals of expression trees, which enables us to modify them without breaking user code. Another effect of the separation is that users always see Rep-types, which make it easy to spot the result type of a query, i.e. the type parameter of Rep.

SIQ only uses LMS' code lifting feature. We don't modify the trees after code lifting and we don't use the information about common subexpressions. Instead SIQ translates queries to Ferry Core as closely as possible and detects common subexpressions later within Relational Algebra expression before generating SQL.

The `implicit def unit` is exposed to user code and automatically converts ordinary Scala typed values into Rep-typed values as required. Within SIQ queries these values count as constants, hence the class name Const.

Changes to LMS

For SIQ, we made a few changes to the core of LMS. The previous sections showed and talked about the original LMS framework. Now we highlight the changes. Some of them might not be general enough for usefulness in other LMS based libraries. They did however prove useful during the development of SIQ.

Restriction of unit The original `implicit def unit` was defined without a type constraint. Any type T could be implicitly converted to a Const[T] a subtype of Rep[T]. During the development of SIQ this lead to unintentional conversions every once in a while. As a consequence we chose to restrict the implicit conversion to selected types. We removed the `implicit def unit` and added implicit conversions for the types supported in SIQ (except Boolean, as explained in section 4.2.2). In particular we defined unitString and unitInt. They can be found in SIQ in traits IModuleBase and ModuleBase, which among other things can be seen in figure 4.2. They serve as intermediate base traits in the SIQ LMS modules.

```
trait IStringOps extends IModuleBase {
  def infix_+( l:Rep[String], r:Rep[String] ): Rep[String]
}
```

```
trait StringOps extends IStringOps with ModuleBase{
  case class Concat(x: Exp[String], y: Exp[String]) extends Def[String]
  def infix_+( l:Exp[String], r:Exp[String] ): Rep[String] = Concat(l, r)
}

trait IModuleBase extends Base{
  trait Implicits{
    implicit def unitInt( v: Int ) : Rep[Int]
    implicit def unitString( v: String ) : Rep[String]
  }
  val implicits : Implicits
  ...
}
trait ModuleBase extends IModuleBase with BaseExp{
  val implicits = new Implicits{
    implicit def unitInt( v: Int ) = Const(v)
    implicit def unitString( v: String ) = Const(v)
  }
  ...
}
```

Easing interactive debugging Expression trees in LMS are composed from Rep-typed nodes. We can see that class Concat has two Exp[String] members. When processing the trees later for translation to Ferry Core, we pattern match them against Const and Sym, the two subclasses of Exp. In case of a Const we can just get out the value and in case of a Sym we have to look up the corresponding Def object in the symbol table. This is no problem when writing code, but when unfolding expression trees in visual debuggers like those provided be Eclipse and IntelliJ, looking up every node in the symbol table manually is tedious.

As a solution, we added a new member def_ to Sym to store the Def object it points to. This allows simply unfolding trees visually without manual lookups.

```
trait Expressions {
      abstract class Exp[+T]
  trait Def[+T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](val id: Int, val def_ : Any) extends Exp[T]
  implicit def toAtom[T](d: Def[T]): Exp[T] = // ... not shown here
}
```

Generating LMS modules In SIQ we noticed that many modules follow the same redundant structure. The implementation trait repeats the method signature of the interface trait and returns an instance of a class which has a unique name representing this particular method. Lifting string concatenation is implemented very similar to lifting of arithmetic operations like -.

```
trait Arith extends Base {
  def infix_-(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
trait ArithExp extends Arith with BaseExp {
  case class Minus(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  def infix_-(x: Exp[Int], y: Exp[Int]) = Minus(x, y)
}
```

Having recognized this structure for most modules, we use code-generation in SIQ to generate them from metadata describing the methods to be lifted (in file dslspec.scala). Currently, only infix-methods with exactly two parameters are supported. The following metadata can be used to generate the LMS modules for arithmetic expressions and string concatenation we have seen before.

```
Module( "String", List(
  Operator( "+", "String", "String", "String" )
))
Module( "Int", List(
  Operator( "-", "Int", "Int", "Int" )
))
```

As a possible future improvement, we could use Scala method signatures as input to the code generator.

Figure 4.2 shows an overview over SIQ's architecture. The left side shows the LMS modules of SIQ and which ones are generated.

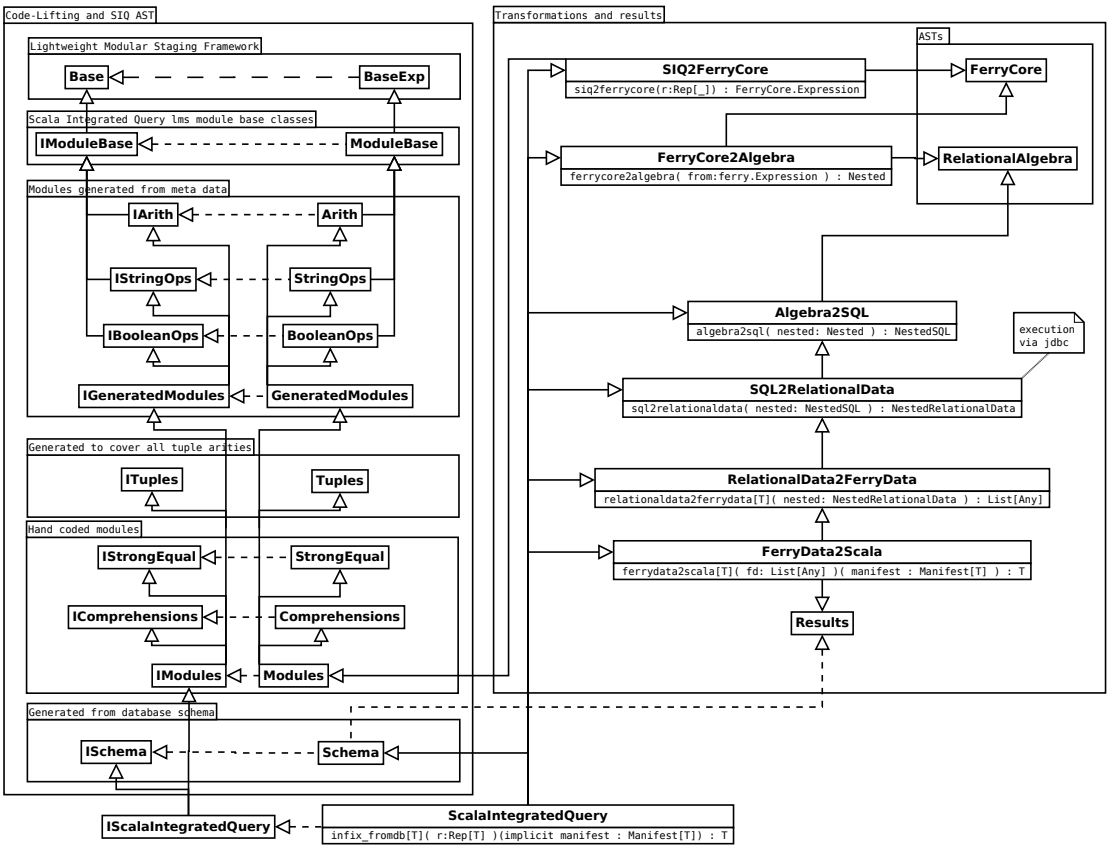


Figure 4.2.: SIQ LMS module architecture

4.2.2. Type safe embedding and lifting

Now we will look at the most interesting LMS modules in SIQ and how type safety and code lifting of different components of the SIQ query language are implemented. We will look at comprehensions, which provide the general structure for expressing queries, at expressions used in projections, filters and ordering, and at some of the operators used in them.

Lifting for-comprehensions

Scala Integrated Query allows expressing queries using Scala for-comprehension syntax or directly using the method calls for-comprehensions are desugared to by the Scala Compiler. In SIQ, lifting of all comprehension related methods is realized in a single LMS module consisting of traits IComprehensions and Comprehensions. We use the common pattern for method extensions in Scala and implicitly convert Rep[Iterable[T]] to instances of sub classes of IGenerator, which provide the available methods map, flatMap, withFilter and orderBy as shown in the following code.

There are three types of Iterables in SIQ, tables, literal tables created from Scala Iterables and comprehensions over other SIQ Iterables. All of them can inherit the same implementation.

orderBy takes a list of functions, which project list elements to Rep-typed expressions, implicitly or explicitly wrapped in Order objects, which determine if it should be translated to ascending or descending order in SQL ORDER BY. SQL restricts ordering to atomic values. We map this restriction to Scala using ValidType as a view bound. Implicit conversions to ValidType only exist for Int, String and Boolean as shown in section 4.2.2.

map takes a projection function and creates a comprehension node as representation in the expression tree.

```
trait IComprehensions extends IModuleBase with ITuples {
  implicit def rep2igenerator[T]( r: Rep[Iterable[T]] ) : IGenerator[T]
  trait IGenerator[+T]{
    def orderBy( orders : (Rep[T] => Order)* ) : Rep[Iterable[T]]
    def map[R]( f : Rep[T] => Rep[R] ) : Rep[Iterable[R]]
    def flatMap[R]( f : Rep[T] => Rep[Iterable[R]] ) : Rep[Iterable[R]]
    def withFilter( f : Rep[T] => Rep[Boolean] ) : Rep[Iterable[T]]
  }

  trait Order
  abstract class IOrderable{
    def asc : Order
    def desc : Order
  }
  implicit def order_enable[T <% ValidType]( r:Rep[T] ) : IOrderable
  implicit def defaultOrder[T <% ValidType]( r:Rep[T] ) : Order
}

trait Comprehensions extends IComprehensions with ModuleBase with Tuples{
  case class LiteralTable[T]( i: Iterable[T] ) extends Generator[T]{ ... }
  abstract class Table[T](...) extends Generator[T]{ ... }

  case class Comprehension[R] (
    var inner : Rep[Iterable[_]],
    val element_ : Rep[R],
```

```
  ) extends Generator[R](element_)

  abstract class Generator[+T]( ... ) extends Def[Iterable[T]] with IGenerator[T]{
    ...
    def map[S]( f : Rep[T] => Rep[S] ) = {
      new Comprehension[S](
        this
        , f( element_references )
      )
    }
    ...
  }
  ...
}
```

Lifting lambdas and other functions

The methods map, flatMap, filter and orderBy take functions as arguments. The code of these functions needs to be lifted in order to translate the contained expressions to SQL. In ordinary Scala, a function passed to the higher-order function map for example would be executed repeatedly for every element in the collection. For the lifting we instead execute it only once passing in a representational Rep-typed value. Of course this requires that the function expects a Rep-type. In case of an in-line anonymous function, this is handled by the Scala compiler's type-inferencer, which automatically types the parameter as a Rep-type. In case of an named function, the parameter type needs to be stated explicitly in the signature. Let's look at an example.

```
employee.map( e => "Name: " + e.name )
```

`employee` is of type Rep[Iterable[Employee]]. The method extensions described earlier provide method `map[R]( f : Rep[T] =>Rep[R] ): Rep[Iterable[R]]`, where T is Employee here. Based on this, the Scala type inferencer types the lambda function as `Rep[Employee] =>Rep[String]`. Accordingly, inside the function `e.name` is of type Rep[String] and accordingly `infix_+( l:Exp[String], r:Exp[String] )` is chosen. As we recognize, Rep-types are viral which allows us to lift the bodies of functions.

The implementation of map applies the projection function once on a representational value to obtain the expression tree. Internally, this value stores a back-reference to the Generator object, so that a reference to the right table or sub query is available during translation to SQL.

Lifting constructors: tuples

SIQ supports tuples. Lifting tuples is not as easy as lifting method calls. The tuple constructor cannot simply be overloaded to return lifted tuples. Instead we let Scala create the tuple and then use implicit conversions to convert tuples to lifted tuples and back. In the following example, in the first call of map, a tuple is lifted because a Rep-type is expected as return value of the projection function. In the second map call it is un-lifted to allow access to its elements.

```
employee.map( e => ( e.id, "Name: " + e.name ) ).map( t => t._1 )
```

To achieve this for tuples of all arities, we provide individual implicits for each arity as shown in the following code of our LMS module responsible for tuples. We used code generation for the implementation.

```
trait ITuples extends IModuleBase{
  implicit def tuple2rep1[T1]( t:Tuple1[Rep[T1]] ) : Rep[Tuple1[T1]]
  implicit def tuple2rep2[T1,T2]( t:Tuple2[Rep[T1],Rep[T2]] ) : Rep[Tuple2[T1,T2]]
  ...
  implicit def rep2tuple1[T1]( t:Rep[Tuple1[T1]] ) : Tuple1[Rep[T1]]
  implicit def rep2tuple2[T1,T2]( t:Rep[Tuple2[T1,T2]] ) : Tuple2[Rep[T1],Rep[T2]]
       ...
  def tuple[T1]( t1:Rep[T1] ) = tuple2rep1( Tuple1(t1) )
  def tuple[T1,T2]( t1:Rep[T1], t2:Rep[T2] ) = tuple2rep2( Tuple2(t1,t2) )
  ...
}
trait Tuples extends ITuples with ModuleBase{
  class LiftedTuple[T]( val p:Product ) extends Def[T]
  case class LiftedTuple1[T1]( t:Tuple1[Rep[T1]] ) extends LiftedTuple[Tuple1[T1]](t)
  implicit def tuple2rep1[T1]( t:Tuple1[Rep[T1]] ) = toAtom( LiftedTuple1( t ) )
  case class LiftedTuple2[T1,T2]( t:Tuple2[Rep[T1],Rep[T2]] ) extends
      LiftedTuple[Tuple2[T1,T2]](t)
  implicit def tuple2rep2[T1,T2]( t:Tuple2[Rep[T1],Rep[T2]] ) = toAtom( LiftedTuple2( t
      ) )
       ...
}
```

This approach comes with a limitation. When a tuple consists of a mix of Rep-types and ordinary Scala types, the implicits do not apply and the query fails at compile time. The following code shows examples of this.

```
employee.map( e => ( e.name, 5 ) )
employee.map( e => ((e.id,e.name), e.name) )
```

The first query fails because e.name is a Rep-type, but 5 is not a Rep-type, so no implicit applies for the tuple. The second example fails, because the first element of the tuple is again a tuple, which is not a Rep-type. We provide a workaround in SIQ to make these cases work. Atomic values can be converted explicitly using method todb. For tuples we offer an alternative constructor called tuple, implemented in trait ITuple shown above. It immediately returns a lifted tuple.

```
employee.map( e => ( e.name, 5.todb ) )
employee.map( e => (tuple(e.id,e.name), e.name) )
```

Lifting of ==

One important operator that needs to be lifted is the equality operator ==. It plays a special role, because it is a member of Scala's global super type Any and takes Any as a parameter. Other operators, like +, we can overload for Rep-types and when a Rep-typed value and an ordinary Scala typed value is provided, the latter is converted to a Rep-type implicitly. == on the other hand is already valid for any combination of parameter types and no implicit conversions will be used. For example in

```
employee.withFilter( e => e.id == 1 )
```

the expression `e.id == 1` normally evaluates to false, as `e.id` is of type Rep[Int] and `1` is of type Int. But of course, we want this comparison to happen in the database, not on the JVM.

Scala Virtualized helps us here as it allows overloading == using an infix method, which takes precedence over the default implementation, when it applies.

Preventing accidents Before we explain the implementation in detail, there is one issue that needs to be discussed. Whiles we overload == method for certain type combinations, for other combinations, we will fall back on the original implementation which returns a Boolean. For example

```
(5 == 6) : Boolean
((x:Rep[_]) == 27) :Rep[Boolean]
```

If the fallback happens unintentionally, for example if a user expects x to be a Rep-type but in fact it is not, a Boolean result occurs. If this happens in an SIQ query, two scenarios are thinkable.

- Scenario 1: There is no implicit conversion Boolean => Rep[Boolean]; only an explicit one. The query fails at compile time because a Rep-type is required.
- Scenario 2: There is an implicit conversion Boolean => Rep[Boolean]. The query compiles and execute without complaint, shadowing the mistake.

The advantage of Scenario 1 is that the mistake leads to a compile time error. But it also means that when someone wants to use Boolean literals, they need to be converted explicitly using method todb.

In Scenario 2 an implicit conversion for Boolean is available. This could be considered more convenient at first glance, as a user can use Boolean values in an SIQ program without having to convert them explicitly. It could however shadow the mentioned mistake.

We believe that here, safety is more important than short-sighted convenience and lower chances for bugs are more important in the long run. This is why SIQ implements Scenario 1.

Strong typing Operators in SQL:1999 are strongly typed [sql08]. The comparison operator = only accepts certain type combinations, otherwise the SQL query will fail. E.g. Numbers can be compared to numbers, but not to Strings. Also, SQL can only compare atomic values, not complete rows. We need to reflect these constraints in SIQ's type safety.

In the implementation we use a view bound, similar to the constraints of orderBy earlier. This way we restrict the allowed types in comparisons to Rep[T] for T being an atomic type: Int, String or Boolean. We also make sure that left hand side and right hand side use the same type T. Two additional overloads allow one side to be an ordinary type and the other side to to be a Rep-type. For other methods like + for String concatenation these cases are already covered by implicit conversions. For == we need to cover them explicitly, so we do not fall back on the default implementation of == on Any.

Another reason why we need the view bound is that Rep[T] is actually Rep[+T] and thus covariant in T. Without the view bound, the compiler could always treat T in both operands

as Any due to covariance and this way a Rep[Int] could be compared to a Rep[String] for example, which would violate our constraints. The view bound solves this by not allowing a global super type.

The following code shows the implementation. Remember that __equal is the magic method name for overloading == in Scala Virtualized.

```
trait IModuleBase extends Base with OverloadHack{
      ...
  // valid types type class
  abstract class ValidType
  implicit def validInt( v:Int ) : ValidType = null
  implicit def validString( v:String ) : ValidType = null
  implicit def validBoolean( v:Boolean ) : ValidType = null
}
trait IStrongEqual extends IModuleBase with OverloadHack {
  def __equal[T <% ValidType](a: Rep[T], b: Rep[T]): Rep[Boolean]
  def __equal[T](a:T, b: Rep[T])(implicit ev: T => ValidType, ev2: T => Rep[T]) :
      Rep[Boolean]
  def __equal[T](a:Rep[T], b: T)(implicit ev: T => ValidType, ev2: T => Rep[T]) :
      Rep[Boolean]
}
trait StrongEqual extends ModuleBase with IStrongEqual {
  case class `==(Any,Any)`(x: Exp[Any], y: Exp[Any]) extends Node[Boolean]( "==" )
  def __equal[T <% ValidType](a: Exp[T], b: Exp[T]): Exp[Boolean] = `==(Any,Any)`(a,b)
  ...
}
```

It interesting to realize what happens at compile time, when comparing a Rep[T] to Rep[S] or S for different types T and S. In this case the Scala compiler still chooses one of our overloads and infers T as Any, but then fails to find an implicit conversion from Any to ValidType, so it fails at compile time.

In cases, where none of the operands is a Rep-type, the compiler falls back to the ordinary implementation of == on Any.

The beauty of LMS, changing == to weak typing The SIQ prototype only implements an SQL:1999 backend using Postgres. But the modularity of LMS makes it possible to adapt the implementation to the specifics of other SQL dialects with a high degree of code re-use. Here we give one example for this.

MySQL for example uses weak typing for ==. Strings can be compared to Ints for example. By replacing the SIQ's LMS module for == by a different implementation we can easily change SIQ's type safety model accordingly. When fully implementing different backends for SIQ in the future, we can use a customized compositions of modules for every backend to combine customization and code re-use.

The following code shows the alternative LMS module for == which implements weak typing. Argument types are still bound to String, Int and Boolean, but differently typed values can be compared with each other.

```
trait IWeakEqual extends IModuleBase with OverloadHack {
  def __equal[T,S](a: Rep[T], b: Rep[S])
    (implicit ev: T => ValidType, ev2: T => Rep[T], ev3: S => ValidType, ev4: S => Rep[S])
    : Rep[Boolean]
  def __equal[T,S](a:T, b: Rep[S])
    (implicit ev: T => ValidType, ev2: T => Rep[T], ev3: S => ValidType, ev4: S => Rep[S])
    : Rep[Boolean]
  def __equal[T,S](a:Rep[T], b: S)
```

```
    (implicit ev: T => ValidType, ev2: T => Rep[T], ev3: S => ValidType, ev4: S => Rep[S])
    : Rep[Boolean]
}
trait WeakEqual extends ModuleBase with IWeakEqual {
      ...
}
```

Putting it all together

Figure 4.2 shows how all of SIQ's LMS modules are combined into one library using inheritance. All interface traits are combined into IScalaIntegratedQuery and all implementation traits into ScalaIntegratedQuery. As we only have SQL:1999 as a backend we only need one composition of modules for now. To use SIQ an instance of ScalaIntegratedQuery needs to be exposed to the user code under the interface IScalaIntegratedQuery. There is a predefined one in the library which can just be imported as seen earlier.

```
object dsl{
  val dsl : IScalaIntegratedQuery = new ScalaIntegratedQuery
}
```

4.2.3. Transformations and ASTs

We described how SIQ lifts queries to expression trees with the help of LMS. We also showed how method fromdb can be used to compile and run queries to obtain the result. In this section we explain the implementation of fromdb, in particular how the steps in chapter 3 are implemented in the prototype.

Calling fromdb triggers a sequence of method calls shown in 4.3, which correspond to the translation steps. Figure 4.2 shows how they are integrated into the SIQ architecture. Each step is implemented in its own trait and method for modularity. To generate a different SQL dialect than SQL:1999 with SIQ, only trait Algebra2SQL has to be replaced by a different implementation. Also we can see from the diagram that only SIQ2FerryCore depends on LMS. The other translation steps could be easily used in a different context like LINQ on Scala.NET.

SIQ to Ferry Core

The first step transforms the SIQ expression trees, created using code lifting into a Ferry Core AST. Figure 4.4 gives an overview over the types of nodes involved in SIQ trees. Figure 4.5 shows nodes involved in the FerryCore AST.

The translation is implemented using a combination of pattern matching and recursion. The provided SIQ expression is matched against the different possible node types and translated according to 3.4. For Sym objects, LMS first requires a lookup in the symbol table to retrieve the expression node. Method t is just a shorthand for method siq2ferryfore.

Implementation type inference for boxing is implemented using attributes of the AST nodes. Every node states its implementation type, either static or inferred from its parameters according to [Sch08] . When translating an SIQ expression, the expected implementation type is provided by the surrounding Ferry Core expression. After the translation,

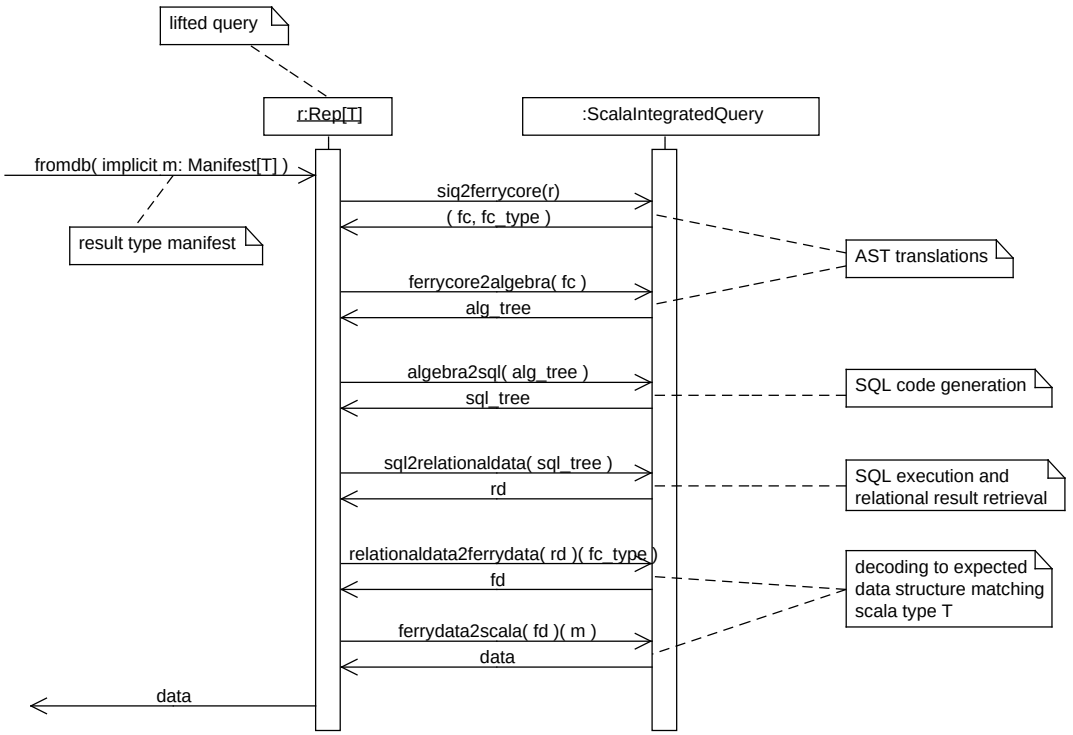


Figure 4.3.: Sequence of actions for compilation, execution and result construction

the expected type is matched against the actual one and the expression is boxed when necessary. Unlike the formal description, this is not implemented as an individual step after translation is done, but on the fly. (The initial expected implementation type is TABLE for Iterables and ROW for tuples. Parameter symboltypes is an implementation detail of boxing. It stores the implementation types of bound variables.)

In the pattern match, from is matched against the two possible options Const and Sym (compare LMS). When it is a Const, simply the Ferry Core AST class representing a literal value is instantiated and returned. When it is a Sym, it is resolved to the corresponding Def subclass, which is then matched against the possible SIQ expression classes. Following the translation rules described in chapter ... a Ferry Core expression is created and returned. When the SIQ expression contains subexpressions, they are translated using method t, providing the necessary implementation type according to (Ferry Thesis). The results are passed to the constructor of the Ferry Core expression class.

```
def siq2ferrycore( from:Rep[_], expected : ImplementationType, symboltypes :
    List[(String,ImplementationType)] = List() ): Expression = {
 def t( from:Rep[_], expected : ImplementationType, symboltypes_ :
     List[(String,ImplementationType)] = symboltypes ) =
   siq2ferrycore(from,expected,symboltypes_)

 val e = from match {
  case Const(value) => ferry.Literal(value)
  case _:Sym[_] => {
```

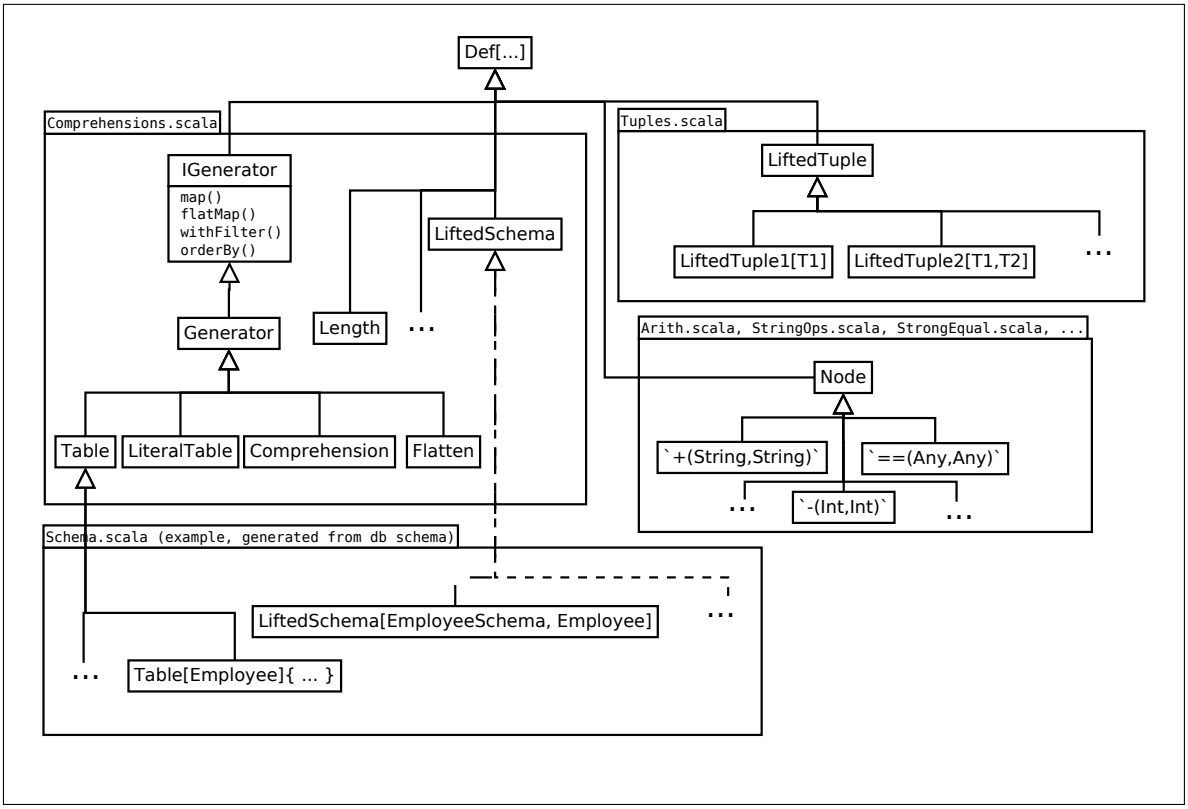


Figure 4.4.: SIQ expression tree nodes

```
    val def_ = rep2def(from)
    def_ match{
      case _:Table[_] => ...
      case _:Comprehension[_] => ...
      case n:Node[_] => ferry.OperatorApplication( n.operator match{
        case "&&" => "and"
        case "||" => "or"
        case "==" => "="
        ...
      }, t(n.x, ROW), t(n.y, ROW) )
      case _:LiftedTuple[_] => ...
      case LiteralTable( values ) => ferry.FerryList( ... )

      case Flatten( list ) => ferry.Concat(
        t(list,TABLE)
      )
      case Length( list ) => ferry.Length(
        t(list,TABLE)
      )
      ...
    }
  }
}
(e.implementation_type, expected) match {
  case (ROW,ROW) => e
  case (TABLE,ROW) => ferry.Box(e)
  case (TABLE,TABLE) => e
```

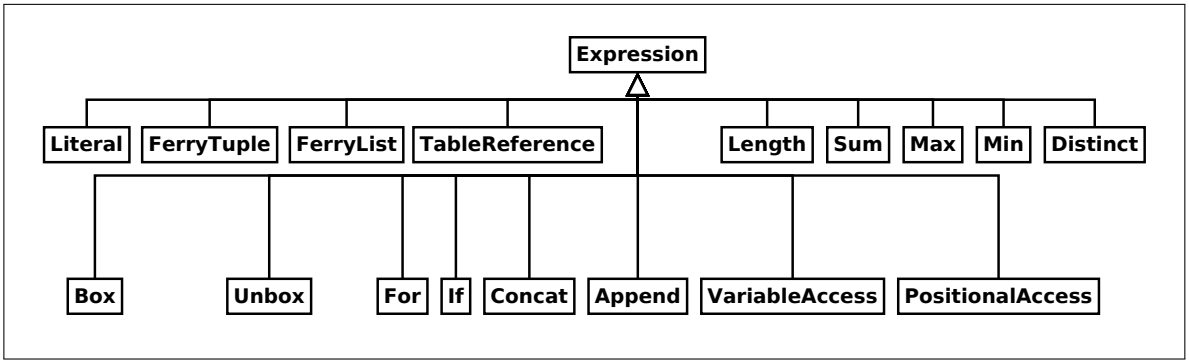


Figure 4.5.: Ferry Core AST

```
    case (ROW,TABLE) => ferry.Unbox(e)
  }
```

Ferry Core to Relational Algebra

The translation to Relational Algebra is implemented very similarly to the previous step using a combination of pattern matching and recursion. Figure 4.6 shows the structure of the target AST.

In correspondence to the translation rules in sections 3.5 and A, ferrycore2algebra takes 3 parameters, an expression from, a loop context loop and an environment of bound variables scope. Method t serves as a short hand for method ferrycore2algebra and sets passes loop context and bound variables through as they are per default, as they only change when entering a comprehension scope. Return type Nested implements table information nodes.

```
trait FerryCore2Algebra extends RelationalAlgebra with Ferry Core{
  val loop_initial = LiteralTable(List(1), List("iter"))

  def ferrycore2algebra( from:ferry.Expression,
                  loop : Relation = loop_initial,
                  scope : List[(String,Nested)] = List()
                 ) : Nested = {

    def t( from:ferry.Expression,
         loop_ : Relation = loop,
         scope_ : List[(String,Nested)] = scope
        ) = ferrycore2algebra(from,loop_,scope_)

    from match {
      case ferry.Box( boxee ) =>
        Nested(
          Attach( 1, "pos", Projection( ("iter", "iter"->"item1"), loop ) ),
          List( "item1" -> t(boxee) )
        )
      case ferry.TableReference(name,columns,keys,order) => ...
      case ferry.For( name, in_, return_, orderBy ) => ...
      case ferry.OperatorApplication( symbol, left, right ) => ...
      case ferry.FerryTuple( head :: tail ) => ...
      case ferry.Concat( lists ) => ...
```

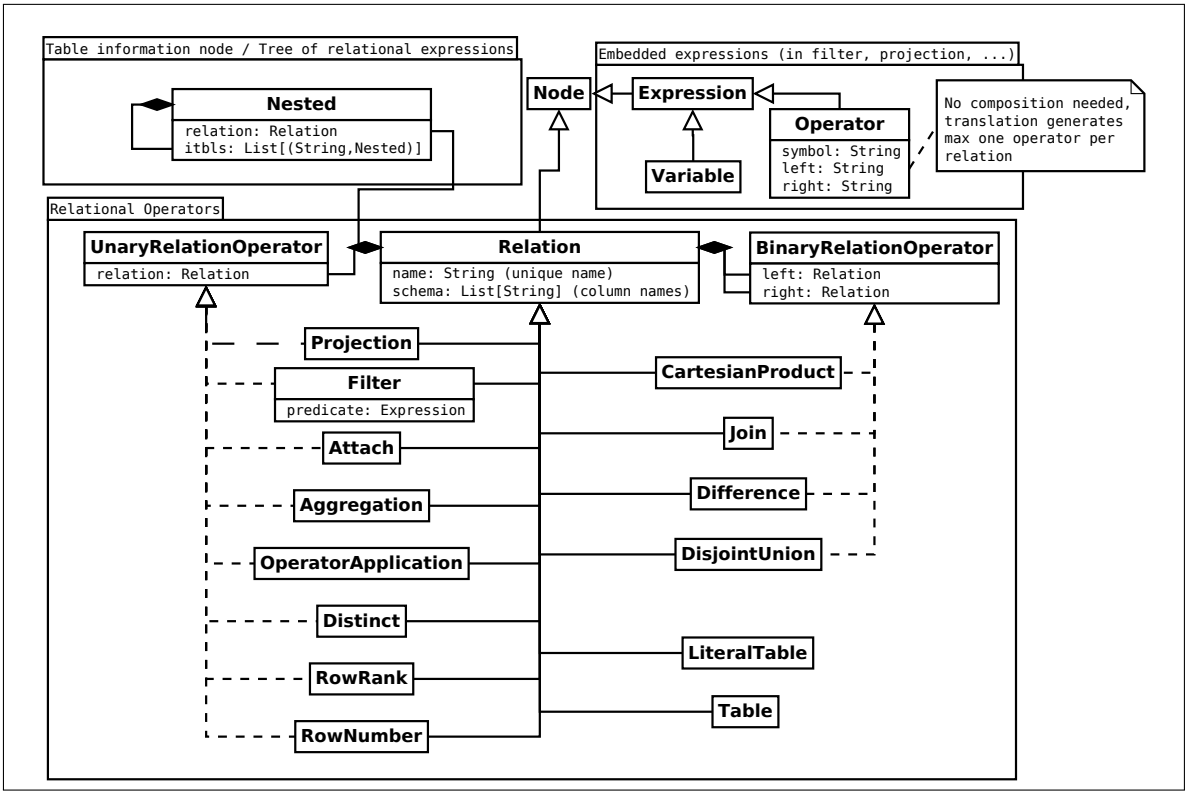


Figure 4.6.: Relational Algebra AST

```
      case ferry.Length( e ) => ...
      ...
    }
  }
}
```

4.2.4. Generating SQL

The next step is SQL code generation. The implementation is very close formal description in section 3.6. Method algebra2sql implements function $\mathcal{X}$, method relation2sql implements function $\mathcal{S}$, method operator2sql implements function $\mathcal{O}$, method expression2sql implements function $\mathcal{P}$ and method linearize_dependencies implements function topsort.

Class NestedSQL shown in figure 4.7 stores the overall result of this step a tree of SQL queries.

```
  def algebra2sql( from:Nested ) : NestedSQL = {
    val sql = relation2sql( from.relation )
    NestedSQL( sql, from.relation.schema, from.itbls.map(_._1) zip
        from.itbls.map(_._2).map(algebra2sql _) )
  }

  def relation2sql( relation:Relation ) = {
    val relations = linearize_dependencies(relation).reverse
```

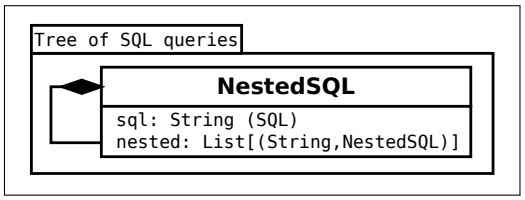


Figure 4.7.: Tree of SQL queries

```
  val sql = "WITH "+
        relations.map( relation => (relation.name + "(%s) AS (%s)") + .format(
          relation.name, relation.schema.mkString(","), operator2sql(relation)
        ).mkString("\n") +
        "\n\n" +
        "SELECT * FROM "+relation.name
    }
def expression2sql( from:Expression ) : String = {
  from match {
    case Variable( name ) => name
    case Operator( symbol, left, right ) => left + " " + symbol + " " + right
  }
}
def operator2sql( from: Relation ) : String =
  from match {
    case p@Projection( _, relation ) => ...
    case Filter( predicate, relation ) => "SELECT * FROM %s WHERE %s".format(
      relation.name,
      expression2sql(predicate)
    )
    case CartesianProduct( left, right ) => ...
    case Join( predicate, left, right ) => "SELECT * FROM " + left.name +","+
        right.name + " WHERE " + expression2sql( predicate )
    case DisjointUnion( left, right ) => "SELECT * FROM " + left.name + " UNION ALL
        SELECT * FROM " + right.name
    case Difference( left, right ) => "SELECT * FROM " + left.name + " EXCEPT ALL
        SELECT * FROM " + right.name
    ...
  }
```

4.2.5. Execution via JDBC

In the next step, method sql2relationaldata traverses the tree of SQL queries, executes every query using JDBC[3] and creates a structurally equivalent tree of relational results, according to definition of function $\mathcal{T}$ in section 3.7.

The database connection is hard coded in the current prototype. Atomic values are retrieved from the database as Strings regardless of the expected Scala type. This situation should certainly be improved in the future.

4.2.6. Constructing the results

Now that we have a tree of results, we can construct the expected result data structure of our overall SIQ program. We traverse the tree of relational results obtained by the previous

[3] http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/

step along-side with the expected Scala type. The nesting structures correspond, because the Ferry mapping split queries according to the type.

The implementation does not exactly correspond to the formal description in section 3.8. We prefer the formally described way, but current implementation is still based on an earlier approach, that divides the process into two steps. The steps are implemented by methods relationaldata2ferrydata and ferrydata2scala which in their tasks roughly correspond functions $\mathcal{Y}$ and $\mathcal{Z}$.

relationaldata2ferrydata only constructs the nesting structure of Iterables is constructed, based on the relational encoding of nesting and the expected Ferry Core type, which roughly corresponds to the expected Scala type, only that nested tuples are flattened. Figure 4.8 shows the data structure used to store the Ferry Core type in the current prototype. The Scala type List[(Int,List[String])] corresponds to the Ferry Core type list(tuple(atomic :: list(atomic) :: Nil)).

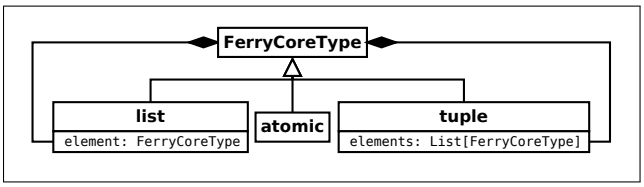


Figure 4.8.: Ferry Core types

The result of relationaldata2ferrydata is a structure of nested lists, in which compared to the expected Scala type, Iterable are represented as Lists, nested tuple and record class structures are also represented as a flat list and atomic values are not yet typed.

```
def relationaldata2ferrydata( nested_data : NestedRelationalData, ferrytype :
    FerryCoreType, iter: Any = "1" ) : Any = {
  val rows = ListMap( nested_data.data.filter(_("iter") == iter).map( ... ) :_* )
  if( nested_data.nested.size == 0 ){
    ferrytype match {
      case a if a == atomic => rows("1")("item1")
      case list(t) if t == atomic => rows.values.map(_.values.toList).flatten.toList
      case list(tuple(_)) => rows.values.map(_.values.toList).toList
      case tuple(_) => rows("1").values.toList
      case _ => throw new Exception()
    }
  }else{
    def handle_tuple_element( e: (FerryCoreType,(String,Any)) ) = e match { case
        (type_,(column,value)) =>
      type_ match {
        case t if t == atomic => value
        case list(_) => relationaldata2ferrydata( nested_data.nested(column), type_,
            value )
      }
    }
    (ferrytype match {
      case tuple(element_types) => element_types zip rows("1") map (handle_tuple_element)
      case list(element_type) => rows.map{ case (pos,row) =>
        element_type match {
          case _:list => relationaldata2ferrydata( nested_data.nested("item1"),
              element_type, row("item1") )
          case tuple(element_types) => ((element_types zip row)
              map(handle_tuple_element)).toList
        }
```

```
      }
    }).toList
  }
 }
}
```

The second step, method ferrydata2scala is responsible constructing nested tuple and record class structures as well as converting atomic values to the expected types and lists to Iterables. The method delegates the task to method getValue, which implements it recursively. getValue receives as first parameter the values at the current nesting level as a mutable stack. In case an atomic value or Iterable is expected the stack contains only one element, which getValue converts to the expected type. If a tuple or record class is expected, the stack contains all values required by the tuple or record class and possibly nested tuples and record classes. It construct the nested structure recursively consuming the component values from the stack one by one. This corresponds to the index $start$ passed through in function $\mathcal{Z}$

For creating instances of record classes getValue uses method createModelInstance defined in the LMS module Schema, which also contains table objects and record classes and is generated from the database schema.

```
def ferrydata2scala[T]( fdata : Any )(m:Manifest[T]) : T = {
  getValue( _stackified(fdata)(m) )( m ).asInstanceOf[A]
}
private def _stackified( x:Any )(m:Manifest[_]) = scala.collection.mutable.Stack(
    (if( isSchema(m) || isTuple(m) ) x.asInstanceOf[List[_]] else List(x)) : _*
)
private def getValue( values:scala.collection.mutable.Stack[Any] )(
     manifest:Manifest[_] ) : Any = {
  (manifest.toString match {
    case "java.lang.String" => values.pop.toString
    case "Int" => values.pop.toString.toDouble.toInt
    case "Double" => values.pop.toString.toDouble
    case _ => {
      if( isTuple(manifest) ){
        val mt = manifest.typeArguments.asInstanceOf[List[Manifest[_]]]
        mt.size match {
          case 1 => Tuple1( getValue(values)(mt(0)) )
          case 2 => Tuple2( getValue(values)(mt(0)), getValue(values)(mt(1)) )
          ...
        }
      } else if( manifest.toString.startsWith("scala.collection.Iterable") ){
        values.pop.asInstanceOf[List[_]].map( x => {
          val nestedType = manifest.typeArguments(0)
          getValue( _stackified(x)(nestedType) )( nestedType )
        })
      } else if(isSchema(manifest)){
        createModelInstance( manifest, values )
      }
    }
  })
}
```

5. Discussion and Future Work

Scala Integrated Query succeeds in its goals to integrate database queries deeply into Scala, providing strong compile time guarantees and greater expressiveness than SQL. It successfully allows easy adaptation to different back ends due to the modularity provided by the Lightweight Modular Staging framework and splits nested queries efficiently into several flat SQL queries based on the Ferry mapping.

While the overall approach is valid and useful, weaknesses in some of the translation steps as well as in the current prototype implementation can be identified and leave room for improvement. This chapter points out the problems and provides an outlook for future solutions.

5.1. Translation steps

The Ferry translation splits a query into a tree of relational expression, each of which we compile to SQL. During our experiments with SIQ using the PostgreSQL DBMS Version 9.0.3 we noticed that the generated SQL queries perform poorly for larger databases. This is not a weakness of our general approach as we can see from other Ferry implementations [Sch08] [GGSW10] [Ulr11] that do not suffer from this problem. But unlike SIQ, these implementations use Pathfinder[1] [BGK+05], a tool that optimizes queries on the level of Relational Algebra and generates efficient SQL queries. It implements optimizations such as Join Graph Isolation [GMR10] and removal of unused columns.

Pathfinder is written in C. Scala programs using SIQ run on the Java Virtual Machine. A dependency on Pathfinder would force an unfortunate constraint on them. Instead it is our plan to port the necessary optimizations from Pathfinder to SIQ in cooperation with the team around Pathfinder and Ferry from University of Tübingen.

We will now explain some of the specific problems that exist in our generated SQL queries.

5.1.1. SQL generation and relational optimization

When translating a relational expression, we translate each relational operator individually and combine them using a WITH expression, as described in section 3.6.1. Appendix sections C.1.1 and C.1.2 show the two generated SQL queries for our running example of employees grouped by workgroups. The individual translation of operators leads to deeply nested queries, which can be easily seen in the SQL code.

Evaluating the SQL queries strictly following their structure would not be very efficient. Repeated scans and sorting of intermediate results would be required. If we are lucky the query optimizer of the used DBMS takes care of this and optimizes the query upfront.

[1] http://www.pathfinder-xquery.org/research/pathfinder/downloads/25-mxq

Figures C.1 and C.2 show EXPLAIN graphs that visualize the execution strategies chosen by PostgreSQL 9.0.3 for our queries. We can see that they are close to the structure of the SQL queries indeed involving repeated scans and sorting. Other DBMS may behave differently.

Sections C.2.1 and C.2.2 show equivalent queries generated by Pathfinder but with relational optimizations disabled (`pf -lIS -o _`)[2]. The corresponding EXPLAIN graphs are shown in figures C.3 and C.4.

We used ferryc, a Ferry implementation developed as part of [Sch08], to generate the necessary input for Pathfinder. As input to ferryc we used our running example query, written in ferryc's input language Ferry.

```
for w in table workgroup(id int,name string) with keys((id)) return (for e in table
    employee(id int, name string, workgroup_id int) with keys((id)) where
    w.id==e.workgroup_id return e )
```

We can see that the SQL queries generated by Pathfinder are already much more compact then those of SIQ and also more efficiently executed by PostgreSQL. The reason is that Pathfinder, does not compile each relational operator individually, but combines several operators into single sub-queries.

Enabling Pathfinder's default optimizations (`pf -lIS`) leads to the SQL queries shown in sections C.3.1 and C.3.2 and the corresponding EXPLAIN graphs in figures C.5 and C.6. The SQL queries are much more compact and the execution strategies involve quite fewer steps, especially removing potentially expensive ones like sort.

Summing up, the Ferry translation of queries involving loop lifting is great for a compositional translation, but can lead to a large number of relational operators. Adding Pathfinder's relational optimizations will be a crucial improvement of SIQ.

5.1.2. Ferry

Ferry computes its own keys for loop lifting and associating values across tables using row number operations. The computation involves sorting, which we can see even in the queries optimized by pathfinder for our running example, even though the source SIQ query does not contain any ordering operation.

However, when writing SQL queries manually instead of SIQ, we can obtain all required data to group employees by workgroup without sorting using the following two queries.

```
SELECT id FROM workgroup
```

```
SELECT * FROM employee
```

Using workgroup.id we can now group the employees manually, even identifying empty groups, just as SIQ does. Instead of an artificial key created involving sorting, we just naturally use the primary key id of table workgroup to associate workgroups with employees. For future work it would be worth investigating to what extend we can change the Ferry translation to use pre-existing keys instead of creating artificial ones for the translation.

[2] We used pf.exe bundled with MonetDB4.

5.2. Scala

Scala's for-comprehensions do not have special syntax for sorting and grouping. [JW07] describes an extension of Haskell's list comprehensions to incorporate these concepts. We should investigate to what extend this approach can be applied to Scala and how useful it would be in the context of SIQ.

The current prototype relies on features in Scala Virtualized. To make SIQ it publicly accessibly these feature must either be ported to the main branch of Scala or SIQ needs to be changed to work without them.

5.3. Prototype implementation

The prototype developed as part of this work has proof-of-concept character. For practice, many things have to be improved, some of which are mentioned in the following.

Currently, SIQ only support PostgreSQL as a backend. In the future, we will support several backends and further explore backends specific type safety using LMS' modularity.

The metadata for generating table objects and records classes should be obtained from reflecting the database schema instead of providing it manually.

The code generation based approach for creating LMS modules should be further explored. Possibly Scaladoc's infrastructure could serve as a basis for generating LMS modules from method signatures.

There might be corner cases, where the current implementation does not enforce type safety effectively, which should investigated more.

The limitations and workarounds of lifting tuples and == described in section 4.2.2 are unfortunate. We should think about alternatives, possibly involving changes to the Scala compiler.

Further limitations of the current implementation include: The database connection is currently hard coded. SIQ only supports a limited set of types and operations, certainly not enough for real world applications. Calling database functions and stored procedures is currently not supported in SIQ.

Many more features would be desirable such as inserts, updates, transactions, ...

6. Related Work

Work on better integrating relational databases with general purpose programming languages has been going on for a long time. Many different approaches overlap in the one or other way. One approach is object-relational mapping. Frameworks, like JPA and Hibernate[1] map the object-oriented model to the relational model, to persist objects and their relationships in relational databases. Another approach, Ferry [Sch08] focuses more on language integration of queries and maps basic data structures like lists and tuples to the relational model. Microsoft LINQ overlaps with both approaches, unifies queries for different backends and integrates them into the .NET languages.

In the Scala world, ScalaQuery [Sca] developed by Stefan Zeiger and Squeryl are stable libraries for type safe database queries. They support multiple SQL dialects and backend DBMS[2]. Like SIQ, ScalaQuery supports Scala's for-comprehension syntax, Squeryl on the other hand uses its own Domain Specific Language (DSL) which looks closer to SQL. Unlike SIQ, ScalaQuery and Squeryl do not support nested result collections, only flat ones. The equivalents to SIQ's record classes and table objects need to be specified manually, while in SIQ they are auto-generated. Inspired by the existence of nested tuples in SIQ, Stefan Zeiger is currently experimenting with adding nested tuple support to ScalaQuery. Both ScalaQuery and Squeryl use a unified type safety model for all backends. SIQ in contrast allows customizing type safety to reflect the specifics of a particular backend, as explained in 4.2.2. A unified model makes it easier to switch an application to another DBMS, while a specific one allows to benefit from the specifics of a particular backend. ScalaQuery and Squeryl support many different practically relevant features not yet included in SIQ, such as Nullable columns and calling database functions.

[SZ09] describes in principle how Scala's for-comprehensions can be used to express database queries, a technique also used in SIQ and ScalaQuery.

Implementations of the Ferry mapping exist for Haskell [GGSW10], Links [Ulr11], Ruby, LINQ [GRS10] and as an independent compiler for the Ferry language ferryc described in [Sch08]. LINQ allows expressing queries with nested results out of the box and the Ferry-based LINQ provider compiles them more efficiently to SQL than Microsoft's LINQ-to-SQL and supports more operators.

Hassan Chafi is working on OptiQL [3] for in-memory queries in Scala. OptiQL conforms to the LINQ API. In LINQ-to-Objects expressions in loop bodies are executed repeatedly. OptiQL identifies loop-invariant expressions and computes them only once, which has some principal similarity to Ferry's accumulation of queries for inner lists into single SQL

[1] http://www.hibernate.org/

[2] ScalaQuery: MySQL, PostgreSQL, H2, Derby, SQLite; Squeryl: MySQL, PostgreSQL, H2, Derby, DB2, MSSQL, Oracle

[3] `https://github.com/stanford-ppl/OptiQL/commits/master/`
`https://github.com/TiarkRompf/virtualization-lms-core/tree/delite-optiql`
`http://www.stanford.edu/class/cs442/lectures_unrestricted/cs442-optiql.pdf`

queries. Since OptiQL runs in-memory queries, nesting is supported natively.

SIQ is neither the only nor the first library built on the Lightweight Modular Staging framework. A series of others exist including

- OptiML [SLB+11], a DSL for machine learning
- Liszt[4] a DSL for building mesh-based solvers of partial differential equations
- StagedSAC a DSL for computations on multi-dimensional arrays

OptiQL is also in the process of migration to LMS. Some of the DSL's already use or are migrating to Delite [BSL+11], a framework built on top of LMS that provides abstractions for parallelization and distribution of programs to heterogeneous hardware like CPUs and GPUs. While SIQ compiles queries and executes them right away, Delite separates compilation and execution. First programs written in Delite DSL's are compiled to optimized code and later run by a separate run time on different hardware.

Re-using an existing compiler and run time code lifting, the technique used by LMS is just one approach to creating DSL's and embedding languages into each other. Very different approaches exist. The Monticore framework [KRV10] generates custom compilers and tools like IDE plugins from language definitions and supports modular development and composition of languages.

[4] http://ppl.stanford.edu/wiki/index.php/Liszt

Appendix

A. Ferry Core to Algebra (cited from [Sch08])

This chapter cites functions and translation rules from [Sch08]. Detailed explanation can be found there.

A.1. Helper functions

`cid(i)`	Generates a column name given a natural number: item1 for i =1, item2 for i = 2, ...
`ord(c)`	Inverse function of cid
$\setminus$	Difference of two sets.
$\lvert\cdot\rvert$	Cardinality of a set.
$[\cdot]_<$	Turns a set of column names into a list of alphabetically sorted column names.
`incr(cols,offset)`	Given a set of column names, increases the index contained in each name. E.g. incr({item1,item2},2) = {item3,item4}.
`keys(m)`	Returns a set of keys of map m.
`incrKeys(m,offset)`	Like incr, but for the keys of a map incrKeys([item1 $\rightarrow$ 1,item2 $\rightarrow$ 2],2) = [item3 $\rightarrow$ 1,item4 $\rightarrow$ 2].
`decrKeys`	The opposite of incrKeys decrKeys([item3 $\rightarrow$ 1,item4 $\rightarrow$ 2],2) = [item1 $\rightarrow$ 1,item2 $\rightarrow$ 2].
`retainByKeys(m,s)`	Given a set of keys s, removes these keys and the values they point to from m.

A.2. Helper rules

$$①q \equiv \#_{\mathsf{item}':\langle \mathsf{iter},\mathsf{ord},\mathsf{pos}\rangle}\left((@_{\mathsf{ord}:1}\,(q_1)) \dot\cup ((@_{\mathsf{ord}:2}\,(q_2)))\right)$$

$$②q' \equiv \pi_{\substack{\mathsf{iter}:\mathsf{item}'',\\ \mathsf{pos},\\ \mathsf{cols}\setminus\mathsf{keys}(itbls_1),\\ \mathsf{keys}(itbls_1):\mathsf{item}'}}\left(q \bowtie_{\substack{\mathsf{ord}=\mathsf{ord}'\wedge\\ \mathsf{iter}=\mathsf{c}'}} \left(\pi_{\substack{\mathsf{ord}':\mathsf{ord},\\ \mathsf{item}'':\mathsf{item}',\\ \mathsf{c}':\mathsf{c}}}\,(q_o)\right)\right)$$

$$\frac{③q \vdash (itbls_1, itbls_2) \overset{itapp}{\Rightarrow} itbls' \qquad ④q_o \vdash \left(\widehat{itbls}, \widetilde{itbls}\right) \overset{itapp}{\Rightarrow} itbls''}{⑤q_o \vdash \left([c \mapsto (q_1, cols, itbls_1)] \uplus \widehat{itbls}, [c \mapsto (q_2, cols, itbls_2)] \uplus \widetilde{itbls}\right) \overset{itapp}{\Rightarrow} \left[c \mapsto \left(q', cols, itbls'\right)\right] \uplus itbls''} \quad \text{(ITAPP-1)}$$

$$\frac{}{q_o \vdash ([\,], [\,]) \overset{itapp}{\Rightarrow} [\,]} \quad \text{(ITAPP-2)}$$

$$\frac{\begin{array}{c}\textcircled{1} q' \equiv \pi_{\mathsf{iter},\mathsf{pos},cols}\left(\left(\pi_{c':c}\left(q_o\right)\right) \bowtail_{\mathsf{c'=iter}} q\right) \\ \textcircled{2} q' \vdash itbls \overset{itsel}{\Mapsto} itbls' \qquad \textcircled{3} q_o \vdash \widehat{itbls} \overset{itsel}{\Mapsto} itbls''\end{array}}{\textcircled{4} q_o \vdash [c \mapsto (q, cols, itbls)] \uplus \widehat{itbls} \overset{itsel}{\Mapsto} \left[c \mapsto \left(q', cols, itbls'\right)\right] \uplus itbls''} \quad \text{(ITSEL-1)}$$

$$\frac{}{q_o \vdash [] \overset{itsel}{\Mapsto} []} \quad \text{(ITSEL-2)}$$

A.3. Translation rules

$$\frac{\Gamma; loop \vdash e \Mapsto ti_e \qquad c = \mathsf{cid}(1) \qquad q_o \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter},c:\mathsf{iter}}\left(loop\right)\right)}{\Gamma; loop \vdash \mathsf{box}(e) \Mapsto \left(q_o, \{c\}, [c \mapsto ti_e]\right)} \quad \text{(BOX)}$$

$$\frac{}{\Gamma; loop \vdash \ell \Mapsto \left(@_{\mathsf{cid}(1):\ell}\left(@_{pos:1}\left(loop\right)\right), \{\mathsf{cid}(1)\}, []\right)} \quad \text{(LITERALEXPR)}$$

$$\frac{\begin{array}{c}* \in \{\mathsf{+}, \mathsf{-}, \mathsf{*}, \mathsf{/}, \mathsf{==}, \mathsf{!=}, <, >, <=, >=, \mathsf{and}, \mathsf{or}\} \\ \Gamma; loop \vdash e_1 \Mapsto (q_{e_1}, \{c\}, []) \qquad \Gamma; loop \vdash e_2 \Mapsto (q_{e_2}, \{c\}, []) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},c:\mathsf{res}}\left(\circledast_{\mathsf{res}:\langle c,c'\rangle}\left(q_{e_1} \bowtail_{\mathsf{iter=iter'}}\left(\pi_{\mathsf{iter'}:\mathsf{iter},c':c}\left(q_{e_2}\right)\right)\right)\right)\end{array}}{\Gamma; loop \vdash e_1 * e_2 \Mapsto (q, \{c\}, [])} \quad \text{(OPAPPEXPR)}$$

$$\frac{\begin{array}{c}\Gamma; loop \vdash e_1 \Mapsto (q_{e_1}, cols_{e_1}, itbls_{e_1}) \qquad \Gamma; loop \vdash (e_2, \ldots, e_n) \Mapsto (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\ cols'_{e_2} = \mathsf{incr}(cols_{e_2}, |cols_{e_1}|) \qquad itbls'_{e_2} = \mathsf{incrKeys}(itbls_{e_2}, |cols_{e_1}|) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},cols_{e_1},cols'_{e_2}}\left(q_{e_1} \bowtail_{\mathsf{iter=iter'}}\left(\pi_{\mathsf{iter'}:\mathsf{iter},[cols'_{e_2}]_<:[cols_{e_2}]_<}\left(q_{e_2}\right)\right)\right)\end{array}}{\Gamma; loop \vdash (e_1, e_2, \ldots, e_n) \Mapsto \left(q, cols_{e_1} \uplus cols'_{e_2}, itbls_{e_1} \uplus itbls'_{e_2}\right)} \quad \text{(TUPLEEXPR-1)}$$

$$\frac{\Gamma; loop \vdash e \Mapsto (q_e, cols_e, itbls_e)}{\Gamma; loop \vdash (e) \Mapsto (q_e, cols_e, itbls_e)} \quad \text{(TUPLEEXPR-2)}$$

$$\frac{\begin{array}{c}\Gamma; loop \vdash e \Mapsto (q_e, cols_e, itbls_e) \qquad c_{old} = \mathsf{cid}(n) \qquad c_{new} = \mathsf{cid}(1) \qquad cols = \{c_{new}\} \\ itbls = \mathsf{decrKeys}(\mathsf{retainByKeys}(itbls_e, \{c_{old}\}), n-1) \qquad q \equiv \pi_{\mathsf{iter},\mathsf{pos},c_{new}:c_{old}}\left(q_e\right)\end{array}}{\Gamma; loop \vdash e.n \Mapsto (q, cols, itbls)} \quad \text{(POSACCEXPR)}$$

$$\frac{\Gamma; loop \vdash e \Mapsto ti}{\Gamma; loop \vdash [e] \Mapsto ti} \quad \text{(LISTEXPR-1)}$$

$$\frac{}{\Gamma; loop \vdash [] \Mapsto \left(\boxed{\begin{array}{|c|}\hline \textbf{iter pos} \\ \hline \quad \\ \hline\end{array}}, \{\}, []\right)} \quad \text{(LISTEXPR-2)}$$

$$\frac{}{[\ldots, v \mapsto ti, \ldots]; loop \vdash v \Mapsto ti} \quad \text{(VAREXPR)}$$

$$[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]; loop \vdash e_1 \Rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1})$$

$$q_v \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter}:\mathsf{inner},cols_{e_1}}\left(\#_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle} q_{e_1}\right)\right)$$

$$loop_v \equiv \pi_{\mathsf{iter}}(q_v) \qquad \mathsf{map} \equiv \pi_{\mathsf{outer}:\mathsf{iter},\mathsf{inner}}\left(\#_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle} q_{e_1}\right)$$

$$\Gamma_v \equiv \left[\ldots, v_i \mapsto \left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},cols_{v_i}}\left(q_{v_i} \bowtie_{\mathsf{iter}=\mathsf{outer}} \mathsf{map}\right), cols_{v_i}, itbls_{v_i}\right), \ldots\right]$$

$$\Gamma_v\left(v \mapsto (q_v, cols_{e_1}, itbls_{e_1})\right); loop_v \vdash e_2 \Rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2})$$

$$\frac{q'_{e_2} \equiv \pi_{\mathsf{iter}:\mathsf{outer},\mathsf{pos}:\mathsf{pos}',cols_{e_2}}\left(\#_{\mathsf{pos}':\langle \mathsf{iter},\mathsf{pos}\rangle/\mathsf{outer}}\left(q_{e_2} \bowtie_{\mathsf{iter}=\mathsf{inner}} \mathsf{map}\right)\right)}{[\ldots, v_i \mapsto (q_{v_i}, cols_{v_i}, itbls_{v_i}), \ldots]; loop \vdash \mathsf{for}\ v\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \Rightarrow \left(q'_{e_2}, cols_{e_2}, itbls_{e_2}\right)} \quad \text{(FOREXPR)}$$

$$\frac{\begin{array}{ll} ① & \Gamma; loop \vdash e_1 \Rightarrow (q_{e_1}, cols, itbls_{e_1}) \quad \Gamma; loop \vdash e_2 \Rightarrow (q_{e_2}, cols, itbls_{e_2}) \\ ② & q \equiv \#_{\mathsf{item}':\langle \mathsf{iter},\mathsf{ord},\mathsf{pos}\rangle}\left(\left(@_{\mathsf{ord}:1}(q_{e_1})\right) \dot{\cup} \left(@_{\mathsf{ord}:2}(q_{e_2})\right)\right) \\ ③ & q' \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}',cols \setminus \mathsf{keys}(itbls_{e_1}),\mathsf{keys}(itbls_{e_1}):\mathsf{item}'}\left(\varrho_{\mathsf{pos}':\langle \mathsf{ord},\mathsf{pos}\rangle}(q)\right) \\ ④ & q \vdash (itbls_{e_1}, itbls_{e_2}) \overset{itapp}{\Rightarrow} itbls' \end{array}}{\Gamma; loop \vdash \mathsf{append}(e_1,e_2) \Rightarrow \left(q', cols, itbls'\right)} \quad \text{(FUN-APPEND)}$$

$$\frac{\begin{array}{c} \Gamma; loop \vdash e \Rightarrow (q_e, \{c\}, [c \mapsto (q_i, cols_i, itbls_i)]) \\ q \equiv \pi_{\mathsf{iter}:\mathsf{iter}',\mathsf{pos}:\mathsf{pos}'',cols_i}\left(\varrho_{\mathsf{pos}'':\langle \mathsf{pos}',\mathsf{pos}\rangle}\left(\left(\pi_{\mathsf{iter}':\mathsf{iter},\mathsf{pos}':\mathsf{pos},c':c}(q_e)\right) \bowtie_{c'=\mathsf{iter}}(q_i)\right)\right) \end{array}}{\Gamma; loop \vdash \mathsf{concat}(e) \Rightarrow (q, cols_i, itbls_i)} \quad \text{(FUN-CONCAT)}$$

$$\frac{\begin{array}{c} \Gamma; loop \vdash e \Rightarrow (q_e, cols_e, itbls_e) \\ q \equiv \mathsf{count}_{\mathsf{cid}(1):\langle\rangle/\mathsf{iter}}(q_e) \qquad q' \equiv @_{pos:1}\left((q) \dot{\cup} \left(@_{\mathsf{cid}(1):0}\left((loop) \setminus (\pi_{\mathsf{iter}}(q))\right)\right)\right) \end{array}}{\Gamma; loop \vdash \mathsf{length}(e) \Rightarrow \left(q', \{\mathsf{cid}(1)\}, []\right)} \quad \text{(FUN-LENGTH)}$$

$$\frac{\begin{array}{c} \mathsf{agg} \in \{\mathsf{avg}, \mathsf{max}, \mathsf{min}, \mathsf{sum}\} \qquad \Gamma; loop \vdash e \Rightarrow (q_e, \{c\}, []) \\ q \equiv \mathsf{agg}_{c:\langle c\rangle/\mathsf{iter}}(q_e) \qquad q' \equiv @_{pos:1}\left((q) \dot{\cup} \left(@_{c:\mathsf{error}}\left((loop) \setminus (\pi_{\mathsf{iter}}(q))\right)\right)\right) \end{array}}{\Gamma; loop \vdash \mathsf{agg}(e) \Rightarrow \left(q', \{c\}, []\right)} \quad \text{(FUN-AGG)}$$

$$\frac{\Gamma; loop \vdash e \Rightarrow (q_e, cols_e, itbls_e) \qquad q \equiv @_{\mathsf{pos}:42}\left(\delta\left(\pi_{\mathsf{iter},cols_e}(q_e)\right)\right)}{\Gamma; loop \vdash \mathsf{distinctValues}(e) \Rightarrow (q, cols_e, itbls_e)} \quad \text{(FUN-DISTINCTVALUES)}$$

B. SIQ prototype folder structure (extract)

- generator-src/
  - dslspec.scala
  - generator.scala
- src/
  - 0_base/
    - lms.scala
    - ModuleBase.scala
  - 0_generated/
    - Arith.scala
    - Boolean.scala
    - Results.scala
    - Schema.scala
    - String.scala
    - Tuples.scala
  - ast/
    - Ferry Core.scala
    - RelationalAlgebra.scala
  - siq/
    - Comprehensions.scala
    - Equal.scala
  - transformations/
    - 2_SIQ2FerryCore.scala
    - 3_FerryCore2Algebra.scala
    - 4_Algebra2SQL.scala
    - 5_SQL2RelationalData.scala
    - 6_RelationalData2FerryData.scala
    - 7_FerryData2Scala.scala
    - RelationalData2Graph.scala
  - templates/

- test-src/
  - tests.scala
- dsl.scala

C. SQL queries for our example

The graphics in this chapter were generated using pgAdmin[1]. The Pathfinder-generated SQL queries were generated using `pf.exe` bundled with MonetDB4 (`http://www.monetdb.org/`).

C.1. Example SQL queries, SIQ

C.1.1. Subquery workgroup

```
WITH

t2(item1,item2) AS
(SELECT workgroup.id AS item1, workgroup.name AS item2 FROM workgroup),

t3(pos,item1,item2) AS
(SELECT ROW_NUMBER() OVER (ORDER BY item1,item2) AS pos, t2.item1,t2.item2 FROM t2),

t1(iter) AS
(VALUES (1)),

t4(iter,pos,item1,item2) AS
(SELECT t1.iter,t3.pos,t3.item1,t3.item2 FROM t1,t3),

t5(inner_,iter,pos,item1,item2) AS
(SELECT ROW_NUMBER() OVER (ORDER BY iter,pos) AS inner_,
    t4.iter,t4.pos,t4.item1,t4.item2 FROM t4),

t9(outer_,inner_) AS
(SELECT t5.iter AS outer_, t5.inner_ AS inner_ FROM t5),

t6(iter,item1,item2) AS
(SELECT t5.inner_ AS iter, t5.item1 AS item1, t5.item2 AS item2 FROM t5),

t7(pos,iter,item1,item2) AS
(SELECT 1 AS pos,t6.iter,t6.item1,t6.item2 FROM t6),

t8(iter) AS
(SELECT t7.iter AS iter FROM t7),

t10(iter,item1) AS
(SELECT t8.iter AS iter, t8.iter AS item1 FROM t8),

t11(pos,iter,item1) AS
(SELECT 1 AS pos,t10.iter,t10.item1 FROM t10),

t51(pos,iter,item1,outer_,inner_) AS
(SELECT * FROM t11,t9 WHERE iter = inner_),

t52(pos_,pos,iter,item1,outer_,inner_) AS
```

[1] `http://www.pgadmin.org/`

```
(SELECT ROW_NUMBER() OVER (PARTITION BY outer_ ORDER BY iter,pos) AS pos_,
    t51.pos,t51.iter,t51.item1,t51.outer_,t51.inner_ FROM t51),

t53(iter,pos,item1) AS
(SELECT t52.outer_ AS iter, t52.pos_ AS pos, t52.item1 AS item1 FROM t52)

SELECT * FROM t53
```

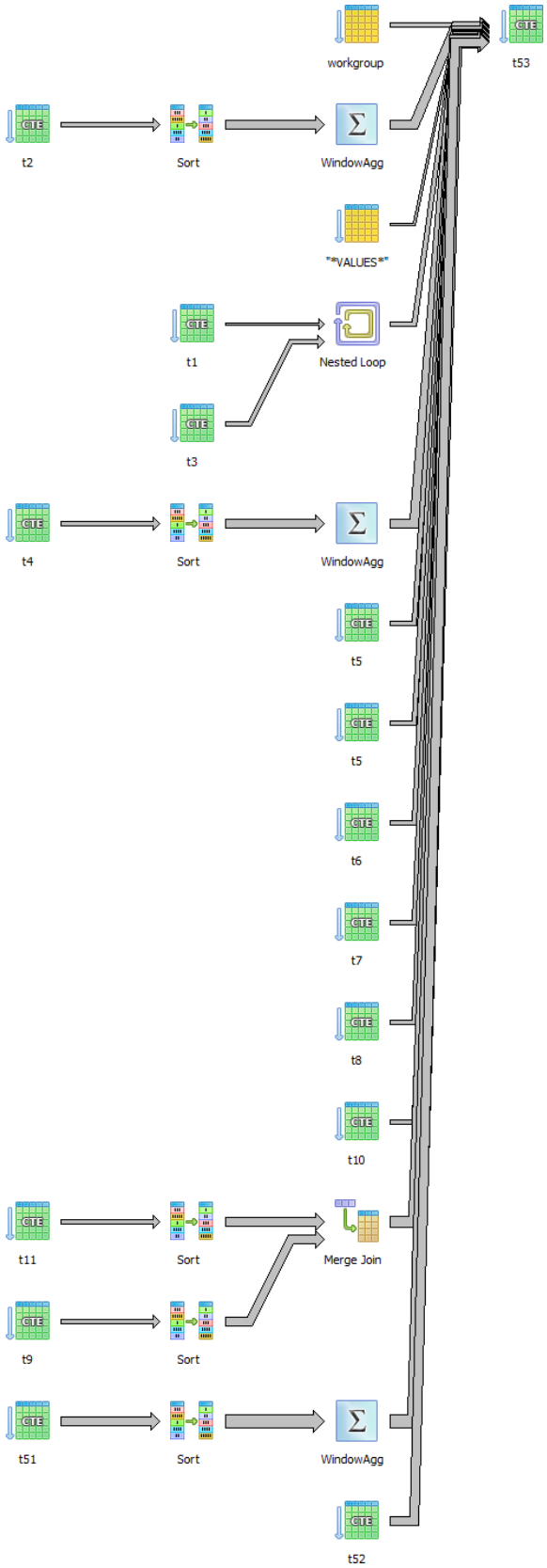


Figure C.1.: PostgreSQL EXPLAIN graph: SIQ Subquery workgroup

C.1.2. Subquery employee

```
WITH

t12(item1,item2,item3) AS
(SELECT employee.id AS item1, employee.name AS item2, employee.workgroup_id AS item3
    FROM employee),

t13(pos,item1,item2,item3) AS
(SELECT ROW_NUMBER() OVER (ORDER BY item1,item2,item3) AS pos,
    t12.item1,t12.item2,t12.item3 FROM t12),

t2(item1,item2) AS
(SELECT workgroup.id AS item1, workgroup.name AS item2 FROM workgroup),

t3(pos,item1,item2) AS
(SELECT ROW_NUMBER() OVER (ORDER BY item1,item2) AS pos, t2.item1,t2.item2 FROM t2),

t1(iter) AS
(VALUES (1)),

t4(iter,pos,item1,item2) AS
(SELECT t1.iter,t3.pos,t3.item1,t3.item2 FROM t1,t3),

t5(inner_,iter,pos,item1,item2) AS
(SELECT ROW_NUMBER() OVER (ORDER BY iter,pos) AS inner_,
    t4.iter,t4.pos,t4.item1,t4.item2 FROM t4),

t6(iter,item1,item2) AS
(SELECT t5.inner_ AS iter, t5.item1 AS item1, t5.item2 AS item2 FROM t5),

t7(pos,iter,item1,item2) AS
(SELECT 1 AS pos,t6.iter,t6.item1,t6.item2 FROM t6),

t8(iter) AS
(SELECT t7.iter AS iter FROM t7),

t14(iter,pos,item1,item2,item3) AS
(SELECT t8.iter,t13.pos,t13.item1,t13.item2,t13.item3 FROM t8,t13),

t15(inner_,iter,pos,item1,item2,item3) AS
(SELECT ROW_NUMBER() OVER (ORDER BY iter,pos) AS inner_,
    t14.iter,t14.pos,t14.item1,t14.item2,t14.item3 FROM t14),

t19(outer_,inner_) AS
(SELECT t15.iter AS outer_, t15.inner_ AS inner_ FROM t15),

t20(pos,iter,item1,item2,outer_,inner_) AS
(SELECT * FROM t7,t19 WHERE iter = outer_),

t21(iter,pos,item1,item2) AS
(SELECT t20.inner_ AS iter, t20.pos AS pos, t20.item1 AS item1, t20.item2 AS item2 FROM
    t20),

t23(iter,pos,item1) AS
(SELECT t21.iter AS iter, t21.pos AS pos, t21.item1 AS item1 FROM t21),

t24(iter_,item2) AS
(SELECT t23.iter AS iter_, t23.item1 AS item2 FROM t23),

t16(iter,item1,item2,item3) AS
(SELECT t15.inner_ AS iter, t15.item1 AS item1, t15.item2 AS item2, t15.item3 AS item3
    FROM t15),
```

```
t17(pos,iter,item1,item2,item3) AS
(SELECT 1 AS pos,t16.iter,t16.item1,t16.item2,t16.item3 FROM t16),

t22(iter,pos,item1) AS
(SELECT t17.iter AS iter, t17.pos AS pos, t17.item3 AS item1 FROM t17),

t25(iter,pos,item1,iter_,item2) AS
(SELECT * FROM t22,t24 WHERE iter = iter_),

t26(iter,pos,item1,iter_,item2,res) AS
(SELECT *,item1 = item2 AS res FROM t25),

t27(iter,pos,item1) AS
(SELECT t26.iter AS iter, t26.pos AS pos, t26.res AS item1 FROM t26),

t28(iter,pos,item1) AS
(SELECT * FROM t27 WHERE item1),

t29(iter) AS
(SELECT t28.iter AS iter FROM t28),

t30(iter_) AS
(SELECT t29.iter AS iter_ FROM t29),

t31(pos,iter,item1,item2,item3,iter_) AS
(SELECT * FROM t17,t30 WHERE iter = iter_),

t32(iter,pos,item1,item2,item3) AS
(SELECT t31.iter AS iter, t31.pos AS pos, t31.item1 AS item1, t31.item2 AS item2,
    t31.item3 AS item3 FROM t31),

t38(iter,pos,item1) AS
(SELECT t32.iter AS iter, t32.pos AS pos, t32.item3 AS item1 FROM t32),

t39(iter_,item2) AS
(SELECT t38.iter AS iter_, t38.item1 AS item2 FROM t38),

t37(iter,pos,item1) AS
(SELECT t32.iter AS iter, t32.pos AS pos, t32.item2 AS item1 FROM t32),

t40(iter,pos,item1,iter_,item2) AS
(SELECT * FROM t37,t39 WHERE iter = iter_),

t41(iter,pos,item1,item2) AS
(SELECT t40.iter AS iter, t40.pos AS pos, t40.item1 AS item1, t40.item2 AS item2 FROM
    t40),

t42(iter_,item2,item3) AS
(SELECT t41.iter AS iter_, t41.item1 AS item2, t41.item2 AS item3 FROM t41),

t36(iter,pos,item1) AS
(SELECT t32.iter AS iter, t32.pos AS pos, t32.item1 AS item1 FROM t32),

t43(iter,pos,item1,iter_,item2,item3) AS
(SELECT * FROM t36,t42 WHERE iter = iter_),

t44(iter,pos,item1,item2,item3) AS
(SELECT t43.iter AS iter, t43.pos AS pos, t43.item1 AS item1, t43.item2 AS item2,
    t43.item3 AS item3 FROM t43),

t45(ord,iter,pos,item1,item2,item3) AS
(SELECT 1 AS ord,t44.iter,t44.pos,t44.item1,t44.item2,t44.item3 FROM t44),

t46(item_,ord,iter,pos,item1,item2,item3) AS
```

```
(SELECT ROW_NUMBER() OVER (ORDER BY iter,ord,pos) AS item_,
    t45.ord,t45.iter,t45.pos,t45.item1,t45.item2,t45.item3 FROM t45),

t47(iter,pos,item1,item2,item3) AS
(SELECT t46.iter AS iter, t46.pos AS pos, t46.item1 AS item1, t46.item2 AS item2,
    t46.item3 AS item3 FROM t46),

t48(iter,pos,item1,item2,item3,outer_,inner_) AS
(SELECT * FROM t47,t19 WHERE iter = inner_),

t49(pos_,iter,pos,item1,item2,item3,outer_,inner_) AS
(SELECT ROW_NUMBER() OVER (PARTITION BY outer_ ORDER BY iter,pos) AS pos_,
    t48.iter,t48.pos,t48.item1,t48.item2,t48.item3,t48.outer_,t48.inner_ FROM t48),

t50(iter,pos,item1,item2,item3) AS
(SELECT t49.outer_ AS iter, t49.pos_ AS pos, t49.item1 AS item1, t49.item2 AS item2,
    t49.item3 AS item3 FROM t49)


SELECT * FROM t50
```

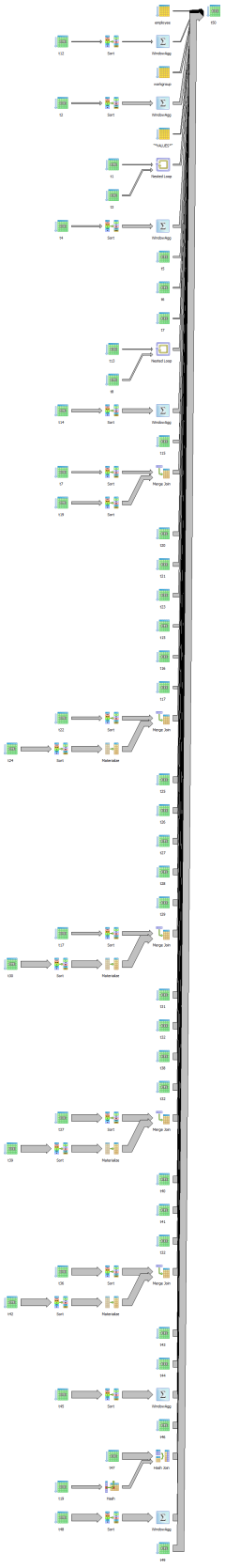


Figure C.2.: PostgreSQL EXPLAIN graph: SIQ Subquery employee

C.2. Example SQL queries, ferryc and Pathfinder, optimizations disabled

C.2.1. Subquery workgroup

```
WITH
-- binding due to rank operator
t0000 (item1_int, item2_str, pos3_nat) AS
  (SELECT a0001.id AS item1_int, a0001.name AS item2_str,
        DENSE_RANK () OVER (ORDER BY a0001.id ASC) AS pos3_nat
    FROM workgroup AS a0001),

-- binding due to rownum operator
t0001 (iter5_nat, item6_int, item7_str, pos8_nat, iter9_nat) AS
  (SELECT a0000.iter4_nat AS iter5_nat, a0002.item1_int AS item6_int,
        a0002.item2_str AS item7_str, a0002.pos3_nat AS pos8_nat,
        ROW_NUMBER () OVER (ORDER BY a0000.iter4_nat ASC, a0002.pos3_nat ASC)
        AS iter9_nat
    FROM (VALUES (1)) AS a0000(iter4_nat),
        t0000 AS a0002)

SELECT a0004.iter5_nat, a0003.iter9_nat
  FROM t0001 AS a0003,
      t0001 AS a0004
 WHERE a0003.iter9_nat = a0004.iter9_nat
 ORDER BY a0004.iter5_nat ASC, a0004.pos8_nat ASC;
```

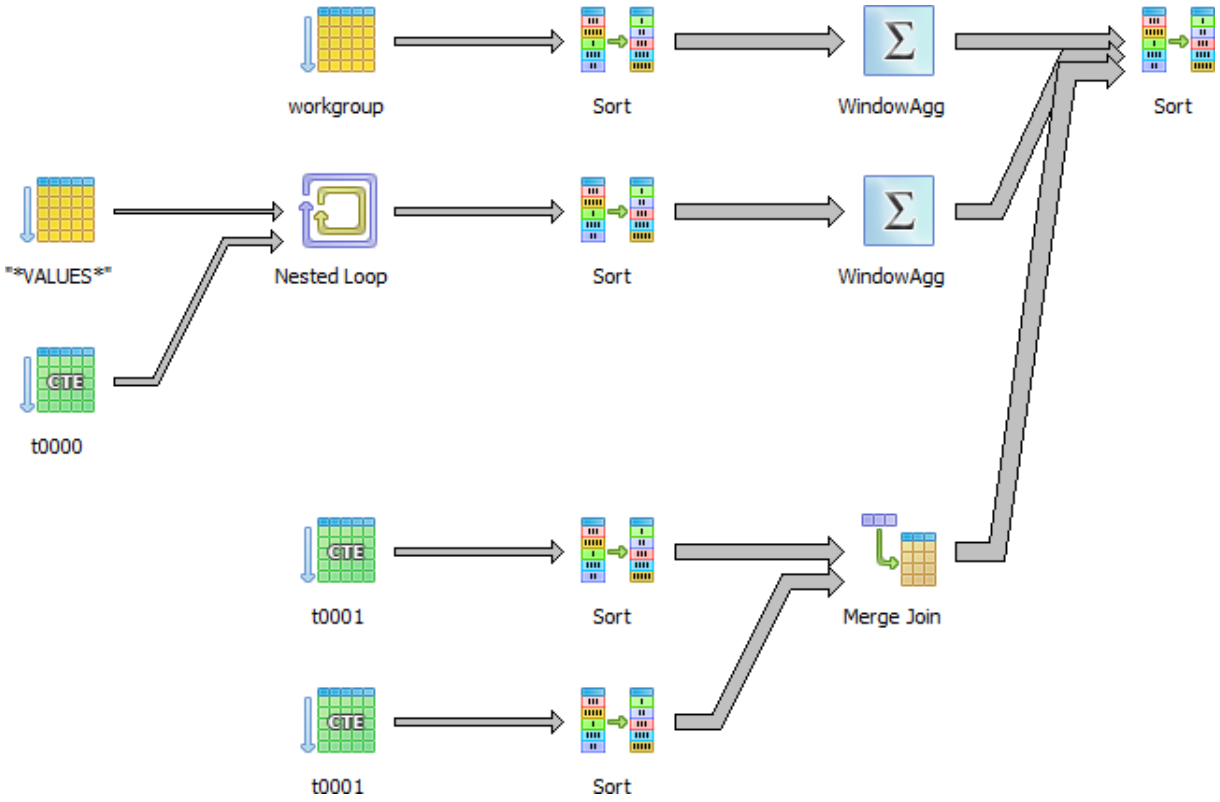


Figure C.3.: PostgreSQL EXPLAIN graph: ferryc and Pathfinder, Subquery workgroup, optimizations disabled

C.2.2. Subquery employee

```
WITH
-- binding due to rank operator
t0000 (item5_int, item6_str, pos7_nat) AS
  (SELECT a0001.id AS item5_int, a0001.name AS item6_str,
        DENSE_RANK () OVER (ORDER BY a0001.id ASC) AS pos7_nat
    FROM workgroup AS a0001),

-- binding due to rownum operator
t0001 (iter9_nat, item10_int, item11_str, pos12_nat, iter13_nat) AS
  (SELECT a0000.iter8_nat AS iter9_nat, a0002.item5_int AS item10_int,
        a0002.item6_str AS item11_str, a0002.pos7_nat AS pos12_nat,
        ROW_NUMBER () OVER (ORDER BY a0000.iter8_nat ASC, a0002.pos7_nat ASC)
        AS iter13_nat
    FROM (VALUES (1)) AS a0000(iter8_nat),
        t0000 AS a0002),

-- binding due to rank operator
t0002 (item1_int, item2_str, item3_int, pos4_nat) AS
  (SELECT a0004.id AS item1_int, a0004.name AS item2_str,
        a0004.workgroup_id AS item3_int,
        DENSE_RANK () OVER (ORDER BY a0004.id ASC) AS pos4_nat
    FROM employee AS a0004),

-- binding due to rownum operator
```

```
t0003 (iter15_nat, item16_int, item17_str, item18_int, pos19_nat,
  iter20_nat) AS
  (SELECT a0003.iter13_nat AS iter15_nat, a0005.item1_int AS item16_int,
        a0005.item2_str AS item17_str, a0005.item3_int AS item18_int,
        a0005.pos4_nat AS pos19_nat,
        ROW_NUMBER () OVER (ORDER BY a0003.iter13_nat ASC, a0005.pos4_nat ASC)
        AS iter20_nat
    FROM t0001 AS a0003,
        t0002 AS a0005)

SELECT a0010.iter15_nat, a0006.item18_int, a0006.item17_str, a0006.item16_int
  FROM t0003 AS a0006,
      t0001 AS a0007,
      t0003 AS a0008,
      t0003 AS a0009,
      t0003 AS a0010
  WHERE a0007.iter13_nat = a0008.iter15_nat
    AND a0008.iter20_nat = a0009.iter20_nat
    AND a0007.item10_int = a0009.item18_int
    AND a0006.iter20_nat = a0008.iter20_nat
    AND a0006.iter20_nat = a0010.iter20_nat
  ORDER BY a0010.iter15_nat ASC, a0010.pos19_nat ASC;
```

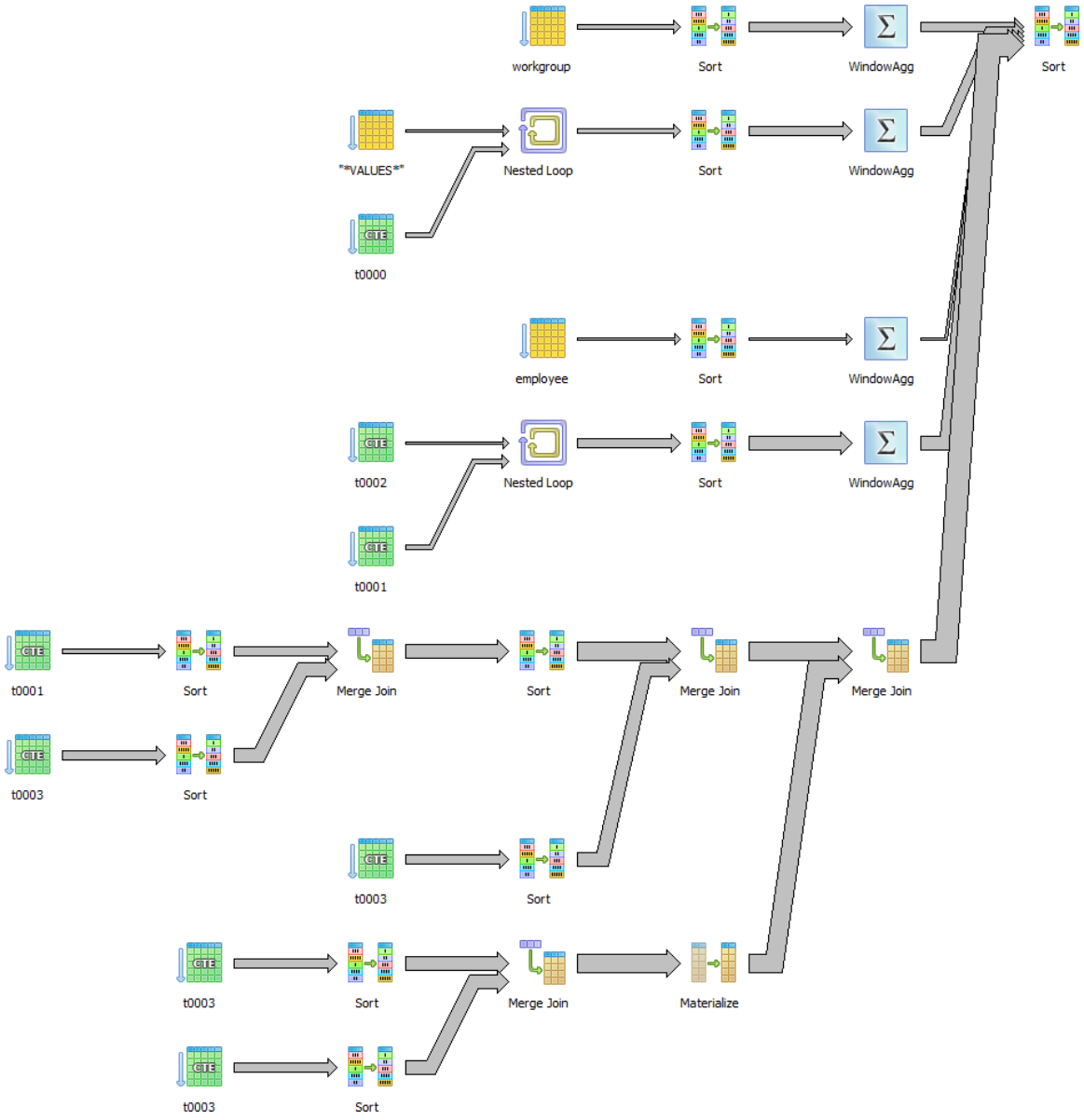


Figure C.4.: PostgreSQL EXPLAIN graph: ferryc and Pathfinder, Subquery employee, optimizations disabled

C.3. Example SQL queries, ferryc and Pathfinder, optimizations enabled

C.3.1. Subquery workgroup

```
SELECT 1 AS iter4_nat,
       ROW_NUMBER () OVER (ORDER BY a0000.id ASC) AS iter3_nat
  FROM workgroup AS a0000
```

```
  ORDER BY a0000.id ASC;
```

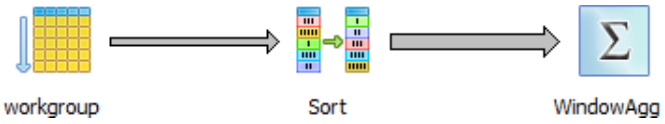


Figure C.5.: PostgreSQL EXPLAIN graph: ferryc and Pathfinder, Subquery workgroup, optimizations ensabled

C.3.2. Subquery employee

```
WITH
-- binding due to rownum operator
t0000 (item4_int, item5_str, iter6_nat) AS
  (SELECT a0000.id AS item4_int, a0000.name AS item5_str,
        ROW_NUMBER () OVER (ORDER BY a0000.id ASC) AS iter6_nat
    FROM workgroup AS a0000)

SELECT a0002.workgroup_id AS item11_int,
       a0002.name AS item10_str,
       a0002.id AS item9_int, a0001.iter6_nat AS iter8_nat
  FROM t0000 AS a0001,
       employee AS a0002
 WHERE a0001.item4_int = a0002.workgroup_id
 ORDER BY a0001.iter6_nat ASC, a0002.id ASC;
```

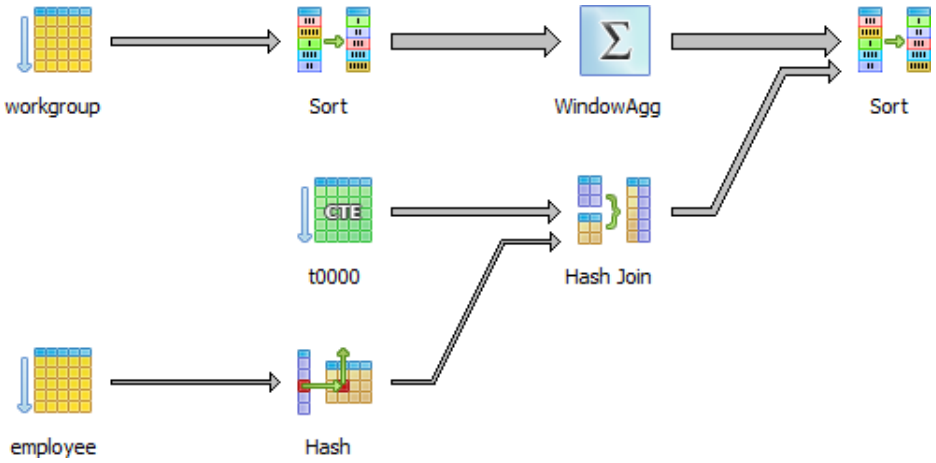
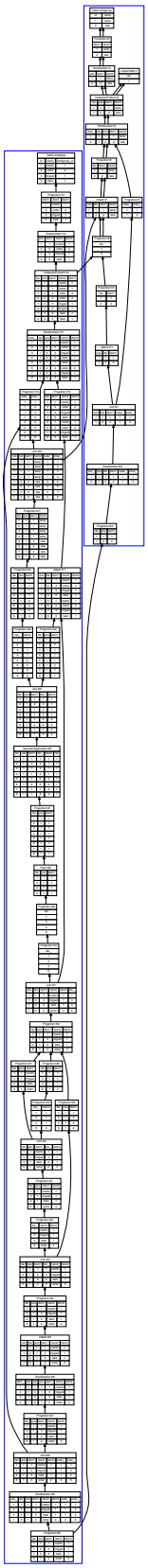


Figure C.6.: PostgreSQL EXPLAIN graph: ferryc and Pathfinder, Subquery employee, optimizations ensabled

D. SIQ example query, intermediate relational results graph



E. CD and project website

The source code of the prototype and Scala Virtualized can be found on the CD or at
`http://code.google.com/p/scala-integrated-query/`
respectively
`https://github.com/TiarkRompf/scala-virtualized`.

The prototype is built with Simple Built Tool (SBT). To execute the tests change to the project folder and run test-run from SBT.

Bibliography

[BGK+05] BONCZ, Peter ; GRUST, Torsten ; KEULEN, Maurice ; MANEGOLD, Stefan ; RITTINGER, Jan ; TEUBNER, Jens: Pathfinder: XQuery-The Relational Way. In: *31st Int'l Conference on Very Large Databases (VLDB)*, 2005

[BSL+11] BROWN, Kevin J. ; SUJEETH, Arvind K. ; LEE, HyoukJoong ; ROMPF, Tiark ; CHAFI, Hassan ; OLUKOTUN, Kunle: A Heterogeneous Parallel Framework for Domain-Specific Languages. In: *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011

[CDM+10] CHAFI, Hassan ; DEVITO, Zach ; MOORS, Adriaan ; ROMPF, Tiark ; SUJEETH, Arvind ; HANRAHAN, Pat ; ODERSKY, Martin ; OLUKOTUN, Kunle: Language Virtualization for Heterogeneous Parallel Computing. 2010. – Forschungsbericht

[GGSW10] GIORGIDZE, George ; GRUST, Torsten ; SCHREIBER, Tom ; WEIJERS, Jeroen: Haskell Boards the Ferry: Database-Supported Program Execution for Haskell. In: *Proceedings of the 22nd Symposium on Implementation and Application of Functional Languages*, 2010

[GIS10] GARCIA, Miguel ; IZMAYLOVA, Anastasia ; SCHUPP, Sibylle: Extending Scala with Database Query Capability. In: *Journal of Object Technology* 9 (2010), Juli, Nr. 4, 45-68. `http://dx.doi.org/10.5381/jot.2010.9.4.a3`. – DOI 10.5381/jot.2010.9.4.a3. – ISSN 1660–1769

[GMR10] GRUST, Torsten ; MAYR, Manuel ; RITTINGER, Jan: Let SQL drive the XQuery workhorse (XQuery join graph isolation). In: *Extending Database Technology*, 2010, S. 147–158

[GMRS09] GRUST, Torsten ; MAYR, Manuel ; RITTINGER, Jan ; SCHREIBER, Tom: FERRY: Database-Supported Program Execution. In: *In SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, 2009, S. 1063–1066

[GRS10] GRUST, Torsten ; RITTINGER, Jan ; SCHREIBER, Tom: Avalanche-Safe LINQ Compilation. In: *Proceedings of the VLDB Endowment, Volume 3, September 2010*, 2010

[GRT08] GRUST, Torsten ; RITTINGER, Jan ; TEUBNER, Jens: Pathfinder: XQuery Off the Relational Shelf. In: *IEEE Data Eng. Bull.* 31 (2008), Nr. 4, S. 7–14

[JW07] JONES, Simon P. ; WADLER, Philip: Comprehensive comprehensions. In: *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–674–5, 61–72

[KRV10] KRAHN, Holger ; RUMPE, Bernhard ; VÖLKEL, Steven: MontiCore: a framework for compositional development of domain specific languages. In: *STTT* 12 (2010), Nr. 5, S. 353–372

[MEW08] MARGUERIE, Fabrice ; EICHERT, Steve ; WOOLEY, Jim: *Linq in action*. Greenwich, CT, USA : Manning Publications Co., 2008. – ISBN 9781933988160

[MS01] MELTON, Jim ; SIMON, Alan R.: *SQL:1999: understanding relational language components*. Morgan Kaufmann, 2001

[Ode11] ODERSKY, Martin: *Scala By Example*. `http://www.scala-lang.org/docu/files/ScalaByExample.pdf`, 2011

[OM09] ODERSKY, Martin ; MOORS, Adriaan: Fighting bit Rot with Types (Experience Report: Scala Collections). In: KANNAN, Ravi (Hrsg.) ; KUMAR, K N. (Hrsg.): *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)* Bd. 4. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009 (Leibniz International Proceedings in Informatics (LIPIcs)). – ISBN 978–3–939897–13–2, 427–451

[RF10] RATTZ, Joseph ; FREEMAN, Adam: *Pro LINQ: Language Integrated Query in C# 2010*. 1st. Berkely, CA, USA : Apress, 2010. – ISBN 1430226536, 9781430226536

[RO10] ROMPF, Tiark ; ODERSKY, Martin: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs / EPFL. 2010. – Forschungsbericht

[Rum95] RUMPE, Bernhard: Gofer Objekt-System – Imperativ Objektorientierte und Funktionale Programmierung in einer Sprache vereint. In: MARGARIA, Tiziana (Hrsg.): *Kolloquium Programmiersprachen und Grundlagen der Programmierung, Adalbert Stifter Haus, Alt Reichenau, 11.10.1995*, 1995

[SBG+10] SCHREIBER, Tom ; BONETTI, Simone ; GRUST, Torsten ; MAYR, Manuel ; RITTINGER, Jan ; TÜBINGEN, Wsi U.: Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider. In: *Proceedings of the VLDB Endowment, Volume 3, September 2010*, 2010

[Sca] *Scala Query*. `http://scalaquery.org/`,

[Sch08] SCHREIBER, Tom: *Translation of List Comprehensions for relational database systems*, Technische Universität München, Diplomarbeit, 2008

[SLB+11] SUJEETH, Arvind ; LEE, HyoukJoong ; BROWN, Kevin ; ROMPF, Tiark ; CHAFI, Hassan ; WU, Michael ; ATREYA, Anand ; ODERSKY, Martin ; OLUKOTUN, Kunle: OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In: GETOOR, Lise (Hrsg.) ; SCHEFFER, Tobias (Hrsg.): *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. New York, NY, USA : ACM, June 2011 (ICML '11). – ISBN 978–1–4503–0619–5, S. 609–616

[sql08] ; ISO / IEC (Veranst.): *ISO/IEC 9075-2:2008 SQL - Part 2: Foundation (SQL/-Foundation), Third edition*. 2008

[SQO06] ; Microsoft Corporation (Veranst.): *The .NET Standard Query Operators*. `http://download.microsoft.com/download/5/8/6/5868081c-68aa-40de-9a45-a3803d8134b8/standard_query_operators.doc`. Version: 2006

[Squ] *Squeryl*. `http://squeryl.org/`,

[SZ09] SPIEWAK, Daniel ; ZHAO, Tian: ScalaQL: Language-Integrated Database Queries for Scala, 2009

[tcp11] *TPC BENCHMARK H*. 2011

[Ulr11] ULRICH, Alexander: *A Ferry-Based Query Backend for the Links Programming Language*, Eberhard Karls Universität Tübingen, Diplomarbeit, 2011

[Wen09a] WEN, Kaichuan: *Language-Integrated Queries in Scala*, Hamburg University of Technology, Diplomarbeit, 2009

[Wen09b] WEN, Kaichuan: Translation of Java-Embedded Database Queries with a Prototype Implementation for LINQ. 2009. – Forschungsbericht