Master Thesis

**Language-Integrated Queries in Scala**

Final Presentation

Kaichuan Wen
http://www.wen-k.com

Supervised by
Prof. Dr. S. Schupp
Prof. Dr. F. Mayer-Lindenberg
Dr. M. Garcia

TUHH, STS

12. Jan 2010
Agenda

1. Motivation & Background
2. Persistence
3. Query
4. Isomorphism
5. Conclusions & Future Work
6. DEMO, Q & A
Where we are...

1 Motivation & Background - Why? What? How?
2 Persistence
3 Query
4 Isomorphism
5 Conclusions & Future Work
6 DEMO, Q & A
Motivation

One round-trip journey

With two kinds of integration

1. **Programming language & Query language**
   - OO paradigm & functional programming
   - Language-integrated query

2. **Object model & Relational model**
   - Through an intermediate language
   - Isomorphic
Involved technologies

- Host programming language: **Scala**
- Functional query language: **LINQ**
- Object-oriented modeling: **EMF**
- Intermediate language: **Ferry**
- Relational modeling: **RDBMS**
Background

**EMF: OO modeling facility (supported subset)**
- Package, subpackage
- Class, interface (abstract and concrete)
- Attribute (of primitive type, multiple and single)
- Reference (multiple and single)

**Ferry: Lightweight intermediate language**
- Lower level - *unaware of OO concepts*
- Types: Tuple, List, Atomic Type - *ensuring 1NF*
  - eg.: (1,2,”a”), ([2,4],[6,8],true), ([ (1,2),(3,4) ], [“a”,”b”])
- Syntax foundation: **for-where-groupby-orderby-return** construct, similar to that of LINQ

Master Thesis: Language-Integrated Queries in Scala  
Kaichuan Wen  6
Background: Resulting Scenario

1. EMF model & its object population as input
2. Persisted into RDBMS
3. LINQ query embedded in Scala triggered on RDBMS
4. Query results returned as OO population

**Ferry**: intermediate language for Step ② and Step ③
Where we are...

1. Motivation & Background
2. Persistence - *Outbound of our journey*
3. Query
4. Isomorphism
5. Conclusions & Future Work
6. DEMO, Q & A
Persistence: phases (Terms will be explained on next slide.)

Phase 1: Persistence of EMF Class Schema
- Translation
- Translation
- Table Creation

Phase 2: Persistence of Object Population
- Row Insertion

Diagram:
- EMF Class Schema → Ferry Types → Tree of Table Information Nodes → Database
- ① Translation
- ② Translation
- ③ Table Creation
- ④ Row Insertion

EMF Model

Master Thesis: Language-Integrated Queries in Scala
Kaichuan Wen
Phase 1: Persistence of EMF class schema

**EMF class schema**

- **Static structure**: classes, interfaces, relationship between them

**Table Information Nodes**

- Another kind of database table schema kept in main memory
- **Ferry-aware**, enabling the translation of LINQ queries into Ferry (more on next slides)
Phase 1: Persistence of EMF class schema

from **EMF class schema** to **Ferry types**:

- Within own class-frame only
- Single attribute (primitive) → Ferry atomic type
- Structural features (through keys and surrogate values):
  - Multiple attributes → Nested list of atomic type
  - Single reference → Nested list of tuple
  - multiple references → Nested list of tuples

From **Ferry types** to **Table Information Nodes**:

- Similar tree-shaped topology
- Nested structures expanded (coming example)

From **Table Information Nodes** to empty tables in RDBMS:

- one table for each class and interface, *plus*
- bridging tables
Phase 1: Persistence of EMF class schema: Example

**Input:** EMF class schema

- **Class:** Person, Employee, Address
- **Interface:** SpeaksEnglish
- **Multiple attribute:** email
- **Reference:** add in Employee
Output: tree of Table Information Nodes

Person
cols(id, name, age, superClass, subClass, superObj, subObj)

Employee
cols(id, email, adress, phone, interfaces, superClass, subClass, superObj, subObj)

Employee_email
cols(iter, pos, item0)

Employee_address
cols(iter, pos, item0)

Address
cols(id, city, street, number)

Employee_interfaces
cols(iter, pos, interface, obj_id, id)

SpeaksEnglish
cols(id, level)
Phase 2: Persistence of EMF object population

- Fill empty tables created in Step ③
- Reflection to retrieve object data
- Row IDs as surrogate values
- Surrogate values to keep track of class hierarchy
Where we are...

1. Motivation & Background
2. Persistence
3. Query - Inbound of our journey
4. Isomorphism
5. Conclusions & Future Work
6. DEMO, Q & A
Query: Phases

- **Step 1 and Step 3**: implemented in this thesis
- Table Information Nodes still used in Step 1
- **Step 2**: implemented at Uni. Tübingen
Query: Translation between query languages

Step ① supports

- All LINQ keywords, LINQ in SQO
- Selected collection operators
- Multi and single attributes and references, also within super-classes and interfaces (up-casting only)

SQO: Standard Query Operators, desugared from textual LINQ, eg.:
- Textual LINQ: from e in Employee where e.age > 20 select e.add
- SQO: Employee.Where(e => e.age > 20).Select(e => e.add)
Query: Translation between query languages

And Step ② (Ferry compiler: **ferryc**, which we don’t have) performs
- Translating Ferry into selected subset of Relational Algebra
- Optimization
Query: Translation between query languages

Finally in Step 3 we get

- Standard SQL:99 query plan ready to be evaluated
- Using keyword **WITH** to combine results of multiple SQL statements
Query: Examples (only snippets shown)

- **LINQ**: from p in Person where p.age > 20 select p.name
- **SQO**: Person.Where( p => p.age > 20 ).Select( p => p.name )
- **Ferry**: let var_2 =
  
  let var_1 =
    table Person ( . . . )
    with keys (id)
    in
    for p in var_1
    where p.age > 20
    return p
  in
  for p in var_2
  return p.name

- **SQL**: WITH (  
  newTab01( . . . ) AS(SELECT * FROM Person),
  newTab02( . . . ) AS(SELECT * FROM newTab01 WHERE newTab01.age > 20),
) SELECT newTab02.name FROM newTab02

- (More examples in DEMO)
Where we are...

1. Motivation & Background
2. Persistence
3. Query
4. Isomorphism - *Why are things correct?*
5. Conclusions & Future Work
6. DEMO, Q & A
In this chapter, we will argue the preservation of isomorphism in both persistence phase and query phase.

Consider Figure 5.1. Given an object model $O$, it is persisted to yield its corresponding relational model $R$. The bijection between $O$ and $R$ implies that a particular $O$ can only be mapped to one $R$, and a particular $R$ can only be mapped from one $O$. We can perform different operations on $O$ and $R$, which in our project are queries $Q_O$ and $Q_R$, respectively. If queries $Q_O$ and $Q_R$ are semantically equivalent, we can conclude that their results $O'$ and $R'$ are also bijective.

In Section 5.1, we will explain the bijection between $O$ and $R$ due to persistence. In Section 5.2, we will discuss about keeping the semantic equivalence between queries $Q_O$ and $Q_R$.

### Bijective

- Each object model $O$ can only map to one relational model $R$
- Each relational model $R$ can only be mapped from one object model $O$

If:

1. $O$ and $R$ are bijective;
2. Query $Q_O$ on $O$ and query $Q_R$ on $R$ are semantically equivalent

We can conclude that:

- Query results $O'$ and $R'$ are also bijective.
Isomorphism: Bijection of models

Why bijective? **Composing** the bijection of:

- **Data types**
  - Primitive types, ignoring different precision
  - Class: gradually factorized into primitive types

- **OO characteristics (attribute, reference, multiplicity of them)**
  - Surrogate values pointing to bridging tables
  - Different storage location of actual data

- **Hierarchy of object population (inheritence, interface, polymorphism)**
  - Unique database table layout
  - Key values and bridging tables for class-frame orientation
Isomorphism: Semantic equivalence of queries

Why are \( Q_O \) and \( Q_R \) semantically equivalent?

- Translation rules obeying semantics
  - List comprehension: foundation of both LINQ and Ferry
- Type checking in translation rules

Example: translation rule for SQO method **Where**

\[
\begin{align*}
v_1 &= \mathbb{V}() \\
T[\text{src}]_{\Gamma} &= \text{src}' \\
T[\text{pred}]_{\Gamma} &= \text{pred}' \\
\Gamma + [v::t_{src}] &= \Gamma' \\
\end{align*}
\]

\[
T[\text{Where(src, v=>pred)}]_{\Gamma} = \begin{cases} 
\text{let } v_1 = \text{src}' \\
\text{in for } v \text{ in } v_1 \\
\text{where } \text{pred}' \\
\text{return } v)_{\Gamma'}
\end{cases}
\]

- **Semantic**: Apply filter \( \text{pred} \) to retrieve items in sequence \( \text{src} \)
- **Typing**: In: sequence \( \text{src} \), boolean \( \text{pred} \); Out: sequence \( \text{src}' \)
Where we are...

1. Motivation & Background
2. Persistence
3. Query
4. Isomorphism
5. Conclusions & Future Work - *Stop and go.*
6. DEMO, Q & A
Conclusion

- Integration of languages
  1. Object-oriented programming language and functional programming language
  2. Programming language and query language
  3. the integration of ① and ②

- Integration of models
  - Principle: via a lower-level intermediate language (Ferry here)

- Isomorphism
  - Guarantee of correctness and wide applicability
  - After a round-trip journey, what sent out comes back
    - uniquely
    - losslessly
Future Work (1)

- Compiler plug-in for Scala
  - More participation of host-language
  - General techniques reported in related works
    - LINQ Expand for Java (Project work by the same author)
    - The Scala Compiler Corner
      http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/

- Extension of Ferry, eg.:
  - Nested tuples
  - More primitive types
Future Work (2)

- Optimization of translation results
  - Ferry Optimizer can also participate
  - Backend-RDBMS-specific optimization

- Queries in other query languages, on other data models
  - Similar techniques applicable
  - *eg.*: XQuery on XML model

- Possible publication
Where we are...

1 Motivation & Background
2 Persistence
3 Query
4 Isomorphism
5 Conclusions & Future Work
6 DEMO, Q & A - What’s clear? What’s not?
Example EMF model:

What is shown?

- Result of persistence
- Queries to retrieve:
  - Fields, including single and multiple attribute and reference
  - Across different class-frame
  - Anonymous type
Q & A

Thanks for your attention!
Your questions are welcome!