A Statically Typed Query Language for Property Graphs

Norbert Tausch¹, Michael Philippsen¹, and Josef Adersberger²

¹University of Erlangen-Nuremberg, Germany, [norbert.tausch|philippsen]@cs.fau.de
²QAware GmbH, Munich, Germany, adersberger@qaware.de
Overview

- Introduction
  - Basic Terms
  - Problem

- The Property Graph Query Language
  - Basic Idea
  - Statically Typed Graph Classes

- Evaluation

- Conclusion
Architecture and Code Graph

UML

```
«interface» IMailBox
```

```
MailBox
```

Source Code

```
class MailBox implements IMailBox {
}
```
Property Graph

- Graph databases often provide an additional API based on a property graph model.

Example:

- **code.class**
  - name = "MailBox"
  - package = "...

- **code.interface**
  - name = "IMailBox"
  - package = "...

- **code.inheritance**

- **code.declaration**
Property Graph

- Advantages:
  - Clear object-oriented graph representation
  - Easy to adapt
  - Easy to extend with new properties
  - Allows for custom graph traversals

- Can represent a wide range of graph types
  - Simple graph, weighted graph, …
- There is no explicit need for dedicated graph classes that represent distinct graph types.

```scala
val adjacentVertices = codeVertices.flatMap(_.outgoingEdges.
                                      filter("type" -> "code.declaration")).
                            map(_.endVertex).
                            filter("type" -> "code.class")
```
Problem

- Current approaches lack a concept for defining statically typed vertex and edge sets.
  - Graph is a set of general purpose vertices (and edges)
    
    ```scala
    val codeGraph = Set[Vertex](vertex1, vertex2, ...)
    ```
  
  - Vertices and edges are represented by generic classes
  - Untyped graphs are used as input and output sets

  - No typed sets => No polymorphism to use and exploit

- Leads to typical programming errors
  - Illegal argument types
  - Wrong calling contexts
### Problem

- **Example – Illegal argument type**

```scala
def adjacentVertices(codeVertices: Set[Vertex]): Set[Vertex] = {
  codeVertices.flatMap(_.outgoingEdges.
    filter("type" -> "code.declaration").
    map(_.endVertex).
    filter("type" -> "code.class")
}
```

- Can lead to data access errors and unexpected result sets

- The developer has to manually ensure legal argument types
  - Before the function call
  - Inside the function

- Inclusion polymorphism – Ensures legal argument types

```scala
def adjacentVertices(codeGraph: CodeGraph): ClassGraph
```
Problem

- Example – Wrong calling context
  - Assume: Two functions for counting lines of code
    - Function 1 works on graphs comprising code vertices
    - Function 2 works on graphs comprising architecture vertices
    - Both functions use different algorithms depending on the input graph type
  
  - How to ensure that the caller chooses the right function?
  - In pure property graphs no compile-time check can prevent a wrong calling context.

- Overloading polymorphism – Ensures the right calling context
  
  ```python
  def countLOC(codeGraph: CodeGraph): Int
  def countLOC(architectureGraph: ArchitectureGraph): Int
  ```
There is a demand for statically typed vertex and edge sets
- Based on the property graph model
- Make polymorphism available for graphs
- Enable compile-time type checks
- Support both generic and concrete graph elements

Idea 1 – We use parameterized graph classes for that.
- Similar to the classical graph definition: G[V, E]
- But: We need some kind of type unions instead

Idea 2 – Vertex and edge filter classes as type parameters
- Filters are representative for the types in a filtered set
- Filtering is a standard task
These ideas are the base of our Property Graph Query Language (ProGQL):

- Internal domain specific language
- Based on Scala
  - Statically typed
  - Object-oriented
  - Functional
  - Runs on the Java VM
ProGQL’s Vertex and Edge Filters

- Parameterized classes: Filter[QVertex], Filter[QEdge]

- Filters are unary functions

  ```scala
  val match: Boolean = filter(aVertex)
  ```

- Generic type filter: Type[T]

  ```scala
  class CodeVertexType extends Type[QVertex]("code.class", "code.interface", "code.method")
  val vertexFilter = new CodeVertexType
  ```

- Concrete type filter: V[T], E[T]

  ```scala
  val architectureVertexFilter = new V[Component] ||
  new V[Interface] ||
  new _V[Package]
  ```
ProGQL's Statically Typed Graph Classes

- Typed graphs are parameterized graph classes

```scala
class Graph[+VF <: QFilter[QVertex], +EF <: QFilter[QEdge]](
  vertexFilter: VF,
  edgeFilter: EF) { … }
```

```scala``val graph = new Graph(new CodeVertexType, new CodeEdgeType)`

- Filters are automatically applied

```scala
val vertices = graph.vertices
val edges    = graph.edges
```

Type of a vertex filter
Type of an edge filter
Object of a vertex filter
Object of an edge filter

Typed Graph
Automatically applied filters
Untyped Sets
Object references
Global Property Graph
### Evaluation

- **Polymorphisms to prevent illegal argument types**

  ```scala
class CodeGraph extends Graph[CodeVertexType, CodeEdgeType]
class CodeGraph2 extends Graph[CodeVertexType, CodeEdgeType]
class CodeGraph3 extends Graph[CodeVertexType, No[QEdge]]
```

- **Inclusion polymorphism**

  ```scala
def inclusion1(graph: CodeGraph) = ...
def inclusion2(graph: Graph[CodeVertexType, CodeEdgeType]) = ...
def inclusion3(graph: Graph[CodeVertexType, _]): CodeGraph3 = ...
```

- **Parametric polymorphism**

  ```scala
def parametric[G <: Graph[CodeVertexType, CodeEdgeType]](graph: G): G = ...
```
Evaluation

- Polymorphism to prevent wrong calling contexts

```scala
class CodeGraph extends Graph[CodeVertexType, CodeEdgeType]
class CodeGraph2 extends Graph[CodeVertexType, CodeEdgeType]
class ArchitectureGraph extends Graph[ArchitectureVertexType, No[QEdge]]
```

- Overloading polymorphism

```scala
def countLOC(graph: ArchitectureGraph): Int = ...
def countLOC(graph: Graph[CodeVertexType, CodeEdgeType]): Int = ...
```

- Coercion polymorphism

```scala
def coercion[G <% CodeGraph](graph: G) = ...

implicit def convertGraph(graph: ArchitectureGraph): CodeGraph = {
  /* Convert graph to a CodeGraph */
}
```
Case Study

- **Target:** Real-world industrial project (45 kLOC)

- **Domain:** Architecture-to-code consistency
  - **Problem:**
    - Source code evolves
    - Usually leads to divergences between architecture and code
  - **Goal:** Improve software quality

- **Task:** Find architectural flaws
  - Search for divergences is a complex task
  - Solution with ProGQL: only about 20 lines of code

- **Result:** Divergences led to the decision to refactor the project
Conclusion

- Typed graphs and polymorphism
  - Prevent illegal argument types
  - Avoid wrong calling contexts
  - Ease the programming of graph queries
  - Enhance readability and maintainability
  - Help avoiding coding errors

- Property Graph Query Language (ProGQL)
  - Scala-based internal domain specific language
  - Enriches the property graph model with typed graphs
  - Enables the use of polymorphism on graphs
  - Helps to better integrate graph databases into OO languages
Short Biography

Norbert Tausch

has studied data and information technology, as well as software engineering before he worked in industry for 10 years in the domain of automation technology as a software engineer and project lead.

Under his advisor, Prof. Michael Philippsen, Norbert ist now a PhD student at the Programming Systems Group at the Computer Science Department of the University Erlangen-Nuremberg, Germany.

His main research interest is the graph-based traceability analysis.