A Statically Typed Query Language for Property Graphs

Norbert Tausch, Michael Philippson
University of Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen, Germany
{norbert.tausch, philippson}@cs.fau.de

Josef Adersberger
QAware GmbH
Aschauer Str. 32, 81549 Munich, Germany
adersberger@qaware.de

ABSTRACT
Applications that work on network-oriented data often use property graph models. Although their graph data is represented by an object-oriented model, current approaches cannot define statically typed vertex and edge sets. Thus, custom graph operations use untyped input and output sets and cannot exploit crucial concepts like polymorphism. Not only do illegal calling contexts or arguments result in runtime errors or unexpected query results, but also the resulting code tends to be error-prone, unclear, and thus hard to maintain. To solve these problems, we extend the property graph model with typed graph classes and open it up to maintain. To solve these problems, we extend the property graph model with typed graph classes and open it up to maintain.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Frameworks

General Terms
Languages

Keywords
Graph traversal, polymorphism, property graph, typed graph

1. INTRODUCTION
A common way to represent network-oriented data in object-oriented applications is by means of a directed, labeled and attributed multi-graph, a so-called property graph (Fig. 1).

This structural graph model can represent a broad range of different graph types [9] and is widely used in graph based systems, e.g., graph databases. At the core of this model are vertex and edge classes. A vertex has a list of incoming and outgoing edges. An edge has a head and a tail vertex. Vertices and edges are generalized as graph elements and have a map of properties (key/value pairs) that store the application-specific data in a semi-structured way.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IDEAS11 2011, September 21-23, Lisbon [Portugal]
Editors: Bernardino, Cruz, Desai
Copyright © 2011 ACM 978-1-4503-0627-0/11/09 ...$10.00.

Norbert Tausch, Michael Philippson
University of Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen, Germany
{norbert.tausch, philippson}@cs.fau.de

Josef Adersberger
QAware GmbH
Aschauer Str. 32, 81549 Munich, Germany
adersberger@qaware.de

The advantage of a property graph model is its object-oriented representation that implements a graph by means of incidence lists. Thus, it is not only simple to adapt this model to other domains just by providing suitable vertex and edge class adapters, but also existing libraries of textbook algorithms, like shortest-path, are instantly available in these domains. In addition, because of the clear interfaces of vertex and edge classes, developing application-specific graph processing is reasonably straightforward—at least at first sight. There is also the same advantage that makes script languages so popular—one can add new properties to vertices and edges on the fly. With an added hibernation of graphs to disk, this flexibility often feels comfortable.

1.1 Problem
Although the property graph model is an object-oriented graph representation, current approaches lack statically typed vertex and edge sets and—as they are a pre-requisite for it—they also do not provide polymorphism on graphs.

Typed Vertex and Edge Sets: Any query language for graphs takes a graph as input and creates a graph as output, i.e., a query transforms sets (of vertices and edges) into other sets. Without types there is no inheritance hierarchy and from an object-oriented perspective all types of vertices and edges can be in those sets. Hence, when processing those sets, it is unknown which properties are defined for each of the elements in the sets and as a consequence, the code needs to include a check in front of every data access. And in set traversal, the code has to be prepared to work with any type of element, since there are no guarantees. With typed sets only certain types of elements can be in a typed set. It is guaranteed that properties exist. Moreover, when processing a typed set, the code does no longer have to expect all types of elements and to prepare against the unexpected. Hence, typed sets make graph processing much simpler.

Polymorphism is a crucial concept of the object-oriented paradigm that can be exploited for graph query languages as soon as there are typed sets. Two flavors of polymorphism are universal polymorphism and ad-hoc polymorphism [2]. For the former, a graph based algorithm exactly denotes its input and output graph type. This is useful if an algorithm only works properly on a distinct graph type (or on all of its sub-types). Algorithms that work differently depending on the input graph type, but should be used homonymously by
the query developer, use ad-hoc polymorphism/overloading. **Coercion** allows for an implicit conversion of graph types in order to fit to the operation’s input type. With polymorphism the compiler can statically enforce that a graph based operation is only invoked on a suitable type. This helps guard against typical programming errors, e.g., wrong calling contexts or illegal argument types.

### 1.2 Basic Idea

Typed sets and polymorphism make programming of graph queries easier, enhance readability and maintainability, and help avoid coding errors. Hence, we extend the property graph model and make typed sets and polymorphism available for property graph based object-oriented applications:

Firstly, vertex and edge classes get additional attributes in order to provide the type information that is necessary for type identification. Our solution works uniformly on both generic and concrete element types. Secondly, we define vertex and edge filter classes that allow for a clear, consistent, and reusable type identification of vertices and edges. Finally, typed graphs are defined by using vertex and edge filters as parameters for statically typed parameterized graph classes. Such graph classes exactly denote their content types and support all mentioned types of polymorphism.

All of this improves quality as the developer can express graph queries in a statically typed way. The resulting code is less error prone since unintended use of graph based operations is detected at compile-time and can no longer lead to runtime errors. Our new **Property Graph Query Language** (ProGQL) combines the flexibility of generic and semi-structured graph elements with the concrete and structured concepts of the object-oriented paradigm. In order to benefit from the advances of current programming languages, we implement ProGQL in the object-oriented and functional language Scala [8] which is based on the Java VM. Since ProGQL is an internal domain specific language, it is easy to extend by adding new classes and methods.

### 1.3 Organization of the Paper

Sect. 2 presents our typed graphs, followed by a case study in Sect. 3 that emphasizes ProGQL’s usability. Sect. 4 covers the related work and Sect. 5 concludes.

### 2. TYPED GRAPHS

Fig. 2 shows our typed graphs and sketches the API. The traits are only used as pure interfaces and do not contain any implementation (a _trait_ can be seen as an interface that also provides partial implementations, similar to abstract classes [8]). The latter is provided by further classes to which we refer when necessary. Three main concepts concerning typed graphs are described in more detail below: First, vertices and edges with additional type information provided by a uniform element interface, second, vertex and edge filters that use this uniform interface, and finally, the typed graph classes themselves.

#### 2.1 Vertices and Edges

As in the basic property graph model, there are vertices and edges defined by the traits **QVertex** and **QEdge** that are sub-traits of **QElement** (see Fig. 2). Application-specific vertices and edges can be derived in different ways, as shown in Fig. 3. Beside the straightforward variants of generic and concrete classes (3a, b), there are also concrete ones built from class and object adapters [4] (3c, d). Regardless of the way the developer creates the application-specific graph elements, it is still necessary to capture the _type_ of the graph elements. Especially in the wrapper-based extension that type is not identical with the subclass. Moreover, the type needs to be captured in a uniform way, i.e., it should not depend on the extension technique used. We achieve this by adding type information in the trait **QElement** that is inherited by every vertex and edge. In contrast to other properties, the type attributes that are shown in Fig. 4 are mandatory and have to be set by all graph elements. Regardless of the extension technique used and the way a developer chooses to add a vertex or edge to the property graph, the attribute _base_ always refers to the concrete _type_ of an element. In cases 3a to 3c it refers to the vertex or edge object itself, whereas in case 3d it refers to the user object that represents the type of the graph element. But obviously the concrete type of the new graph element is unknown in the general **QElement**. The abstract type member _B_ helps as it has to be specified by all implementing user classes and exactly denotes the concrete vertex or edge types. The additional attribute _typeName_ is a string representation of a type and is mainly needed for generic elements (case 3a), but the string can also be used in concrete elements (cases 3b and 3c).

#### 2.2 Vertex and Edge Filters

Filtering is a standard task of graph processing and for partitioning vertex and edge sets into sub-sets. Technically, filters are unary functions that take a vertex or an edge as their argument and return a boolean value signaling whether or not the graph element matches the filter criterion. Although arbitrary criteria are possible, we focus on type filters that pass an element _if_ it is of a certain type (or types). See Sect. 2.2.2

![Figure 2: Schematic overview of the ProGQL API.](image)

![Figure 3: Four ways to implement vertex classes.](image)

![Figure 4: Trait QElement holds extra type information. A QFilter is a uniform interface for all filters.](image)
for other types of filters. Due to its frequent use, a filter must be declared in a clear, consistent, and reusable way. In ProGQL all filters implement the parameterized trait QFilter (API in Fig. 4). Vertex and edge filters are established by applying the traits QVertex and QEdge as type parameters to it. Thus, for example, the type QFilter[QVertex] is the super-type of all vertex filters. The core filter function itself is the abstract method apply(T) that has to be provided by all concrete filter classes. Then, for example a vertex can be filtered by the following statement that implicitly calls the apply method: vertexFilter(vertex). In order to build complex filters, logical operators (or), && (and), and ! (not) can be used to combine filters.

2.2.1 Type Filters
Type filters use the type information in QElement to make the filtering decision. ProGQL’s flavors of type filters will be explained below. Table 1 shows their syntax and examples.

**Generic Type Filter:** A generic type filter uses the String attribute typeName of vertices and edges for the filtering decision. The names of acceptable types (list of strings) are given to the filter at the time of its construction. The strings are used for comparisons when the filter is applied.

**Concrete Type Filter:** Concrete type filters use the base attribute of elements in order to decide whether an element matches. There are two kinds: concrete type filters match the given type and all its sub-types; strict concrete type filters match the given type only and do not consider inheritance. In contrast to generic type filters, the argument of such a filter is a type provided as a type parameter.

**Concrete Type Union Filter:** As it is cumbersome to combine unary concrete type filters by means of logical operators, we also provide concrete type filters for type unions that accept two type parameters and are then composable. ProGQL’s syntactic sugar is ++ for a vertex filter and +++ for an edge filter. For example new ++[A, ++[B, C]] is a type union filter that matches three vertex types.

2.2.2 Other Filters
Beside type filters there are other types of filters, for example property and edge direction filters. A property filter (provided in class Property) tests if a vertex or an edge has a property or if it has a distinct value. An edge direction filter checks if an edge is outgoing from or incoming to a vertex. ProGQL provides predefined classes In and Out that both expect an additional edge filter as their constructor argument. To test the direction the filter needs a corresponding vertex as context. The second binary apply function in Fig. 4 uses the array parameter for that purpose.

2.2.3 Named Filters
Filters can also be named. Below a concrete filter gets a unique name that can be used instead of a vertex filter: class MyFilter extends Type[QVertex]("a", "b", "c") An instance of it is simply created by the statement new MyFilter. Logical filter combinations can be expressed and named by means of a predefined wrapper for filters that expects another QFilter in its constructor argument: class MyFilter extends FilterWrapper(QType[QVertex]("a", "b", "c") || (A ++ B ++ C))

The example defines a named filter that matches all vertices that either have one of the given type names or are of one of the listed concrete types.

```java
1 val graph1 = new Graph(
2    Type[QVertex]("code.class", "code.interface",
3                  "code.method", "code.field"),
4    Type[QEdge]("code.nest", "code.declaration",
5                  "code.inheritance", "code.field&access"),
6    class CodeVT extends Type[QVertex](...)
7    class CodeET extends Type[QEdge](...)
9 val graph2 = new Graph(new CodeVT, new CodeET)
10 val graph3 = new Graph(CodeVT, CodeET)
```

Figure 5: Examples of graph instantiations using anonymous and named filters.

### 2.3 Typed Graph Classes
We now illustrate our typed graphs that are based on parameterized graph classes and that are amenable for several flavors of polymorphism. We also show how typed graphs can be used for a declarative definition of graph traversals.

2.3.1 Parameterized Graph Classes
Often a typed graph class is understood as a parameterized graph class like class Graph[V, E], where V and E are vertex and edge classes. But in order to define graphs that can hold different types of vertices and edges (not just V and E and their respective sub-types and not just concrete but also generic element types), type unions are needed. However, ProGQL does not add type unions as such. Instead, it uses type filters as they are more expressive than type unions and also lead to a lazy on-demand evaluation. ProGQL provides typed graphs by using the trait Graph (see Fig. 2). It defines a graph interface using both a vertex and an edge filter as type parameters. Class Graph is a generic implementation that expects both filters as constructor arguments from which the compiler automatically infers the corresponding type parameters. Fig. 5 shows how a typed graph can be instantiated using filters as type parameters. The graph in line 1 uses anonymous filters. The example’s domain is a code model containing class and interface artifacts that all can have nested fields and methods (see Fig. 9 for details). The value graph1 refers to a graph of the automatically inferred type Graph[Type[QVertex], Type[QEdge]]. Lines 7-10 show the same example with named filters. Line 9 passes the type filters to the graph as constructor arguments, whereas line 10 uses type parameters. In the latter case, the graph class uses reflection to instantiate the necessary filter objects at runtime and passes them to the graph’s constructor automatically. Both graph2 and graph3 refer to graphs of type Graph[CodeVT, CodeET].

**Figure 6:** Typed graphs are views on other graphs.

**Graphs are views:** To avoid time consuming copy operations when creating sub-graphs or when converting graph types, we use a view concept that is more efficient. Fig. 6 illustrates the idea. The base of a view hierarchy – and therefore of a graph – is a global property graph that holds all the vertices and edges. This network is either
Table 1: Filters for type identification, the respective filter classes, and instantiation examples.

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Filter Class</th>
<th>Edge</th>
<th>Standard Notation</th>
<th>Instantiation Examples</th>
<th>Shorter Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic type</td>
<td>Type[QVertex]</td>
<td>Type[QEdge]</td>
<td>new Type[QVertex](&quot;a&quot;, &quot;b&quot;, &quot;c&quot;)</td>
<td>Type[QVertex](&quot;a&quot;, &quot;b&quot;, &quot;c&quot;)</td>
<td></td>
</tr>
<tr>
<td>Concrete type union</td>
<td>++[T1, T2]</td>
<td>+[T1, T2]</td>
<td>new ++[A, ++[B, C]]</td>
<td>new ++[A, ++B ++ C]</td>
<td></td>
</tr>
</tbody>
</table>

located in memory or it may be provided by a graph database. To reduce the number of vertices and edges that have to be handled in graph-based operations, each graph is based on an untyped base graph that contains a sub-set of all available vertices and the corresponding edges. Conceptually, an untyped base graph itself is also a special implementation of a QGraph using the filters Every[QVertex] and Every[QEdge]. The predefined filter Every simply matches all graph elements. Filter produce typed graphs from this untyped base graph. The view contains only the vertices and edges that match the filters’ criteria. As previously defined, a typed graph comprises of a vertex filter and an edge filter that both are automatically applied to such an underlying so-called pool graph on demand, e.g. when vertices or edges are accessed:

typedGraph.vertices; typedGraph.edges

Filters can also restrict the selected elements of typed graphs. This builds a hierarchy of typed graphs. Graphs are refined from other typed graphs, but at the base of each such graph hierarchy there is always an untyped base graph that holds all vertices and edges. Graphs are strictly layered. Each graph can directly access its directly underlying graph using the pool attribute. It can also access the base graph using a base attribute:

typedGraph.pool.vertices; typedGraph.base

As a typed graph is only a view, new vertices are always added to the underlying base graph, even if the vertex type does not fit to the typed graph. Otherwise wrongly typed vertices would have to be refused resulting in some kind of exception to be handled by the application code.

Casts and Refinements: To trim graphs to the signature of graph operations, as in all statically typed languages, explicit cast and refinement operations are needed. A cast operation is denoted by a to-operator that is available in all QGraph based classes: graph.to[NewGraphType]. This creates a new graph of the type NewGraphType and fills the vertices of the original graph into it. Hence, a cast operation is a one-time conversion and is based on a copying process. In contrast, a refinement is a continuous conversion without any copying: graph.refine[NewGraphType]. The new graph of type NewGraphType uses the original graph as a pool graph.

2.3.2 Polymorphism

Based on statically typed graph classes, developers can use polymorphism to strictly denote input types and output types of their custom graph processing operations. For the description see Fig. 7. Lines 2 to 4 define three named graphs. The class CodeGraph extends the generic class Graph and applies named filters as type parameters to it. While CodeGraph and CodeGraph2 are defined identically, CG3 uses a different edge set provided by the filter No[QEdge] which is a predefined filter that simply does not match any edges at all. Lines 5-14 demonstrate various flavors of polymorphism.

Inclusion polymorphism is used for operations that only work with one distinct graph type (or with any of its sub-types). Method inclusion1 in line 5 only accepts instances of CodeGraph or sub-classes of it, whereas inclusion2 accepts objects of both CodeGraph and CodeGraph2, but not CG3 due to the different edge type set. Only inclusion3 accepts all three code graphs. As denoted by the wildcard \_\_, it accepts all edge filters. Line 7 shows that inclusion polymorphism works with output graphs as well: inclusion3 returns a CG3 or a sub-class of it.

Parametric polymorphism is similar to inclusion polymorphism but works with type parameters instead of fixed argument types. The method parametric in line 8 accepts all graph types that match the given upper bound QGraph[\_\_\_]. The output type is fixed by the same type parameter.

Overloading ad-hoc polymorphism is used by graph algorithms whose implementations vary depending on the given input types and are thus coded in different methods. The method overloading is defined twice in Fig. 7, but with different argument types. In line 10 it only accepts graphs of type CG3 whereas in line 11 it accepts CodeGraph and CodeGraph2 but not CG3.

Coercion: Often an operation only works for a certain type but it can also be used for other graph types if there are ways to convert one type into the other, i.e., if there is coercion polymorphism. For example the parameter of method coercion in line 12 accepts any argument whose type can be implicitly converted to a CodeGraph. Scala provides this feature by using a view bound that is denoted by G <: CodeGraph. Beside a CodeGraph, which does not have to be converted, also a CG3 can be applied as there is an implicit converter from CG3 to a CodeGraph in scope (line 13).

2.3.3 Graph Traversals

ProGQL’s graph queries are always based on graph traver-
sals that can either be implemented manually or that make use of a predefined traversal API. This is advantageous as repetitive coding chores can be avoided. The API uses a novel declarative syntax that is purely based on filters and typed graphs that allows for the definition of graph traversals in a clear, reusable, and concise way. A graph traversal is defined by the trait `QTraversal` (see Fig. 2) that is parameterized with a target graph `T`. The result of a traversal is a typed graph constructed during the traversal process: `val graph = source >> BFSTraversal[CodeGraph]`

This example is a breadth-first traversal, started by the method `>>` of a graph (source) that accepts a typed traversal that returns a typed graph. Fig. 8 shows more parts of graph traversals that are explained below:

**Figure 8: Parts of a typed graph traversal.**

**Path:** Starting from each source vertex, the traversal path that is followed is described by a vertex and an edge filter. It reaches those vertices that are reachable over edges and vertices that match the given filters. Above there is no explicit filter, hence all graph elements are traversed. A traversal can be refined by means of `over`:

```
val t = BFSTraversal[CodeGraph]
val graph = source >> t.over(vFilter, eFilter)
class PathGraph extends Graph(vFilter, eFilter)
val graph2 = source >> t.over(PathGraph)
```

In addition to type filters, property filters, and direction filters, there are additional filters that can actively react on the state of a traversal and therefore can fully control it. An example is the depth filter:

```
val graph = source >> t.refinePath(Depth(2))
```

This depth filter refines the path in order to stop the traversal at a maximum depth of 2 from each source vertex. Developers can create filters to fully control the traversal process and the path in which a graph is traversed.

**Result:** The traversal function visits each vertex on the path and checks with the filter of the target graph whether or not to put it into the result set. This is illustrated in Fig. 8 as an intersection between path and result. This is because often not all vertices that match the filter of the target graph are really required, but only a sub-set of it is desired. Therefore, `ProGQL` provides a way to refine the result set without changing the target graph type: `t.refineResult(Type[QVertex]("code.class"))`. In this case the target graph is of type `CodeGraph`. But from all vertices only those are in the target view that match the extra refinement. Hence, the developer has full control over which elements actually make it into the target graph.

### 3. CASE STUDY

To illustrate the use of `ProGQL`, we now provide a case study. The example domain is architecture-to-code consistency, based on so-called reflection models [7]. Fig. 9 depicts the schema of the corresponding graph and its three main parts. First, the **architectural model** is based on a `UML` model of an industrial project provided by a software architect. It consists of components and interfaces. Interfaces are either provided or required by components. In the given global property graph architectural artifact and link types are represented by concrete vertex and edge classes. Second, the **code model** holds types (classes and interfaces) and their nested methods and fields. Types can be used to declare other types, fields, or methods. A type can also be used to declare an inheritance hierarchy and to nest methods that can access fields or that call other methods. In the graph representation code artifacts and links are represented by generic vertices and edges. We have extracted this graph from the project’s Java code, provided by the client’s software developers. Finally, both models are correlated by **architecture-to-code traceability** links that denote the relationships between types and architectural artifacts. These edges are a result of an automated reflexion analysis and human inspection.

It was `ProGQL`’s task to automatically find so-called divergences between architecture and code. A divergence for instance emerges when software developers introduce code dependencies that have not been modelled by the software.

**Figure 9: Graph of an architectural model, a code model, and one for architecture-to-code traceability.**

**Figure 10: `ProGQL` example of identifying divergences between architecture and code.**

```scala
| class ArchGraph extends Graph(new (Component ++
| Interface), Out(new (ProvideInterface ++
| RequireInterface)))
| class ReflexionModel {
|   class Path1 extends Graph(Every[QVertex], Out(
|     new CodeET))
|   class Path2 extends Graph(Every[QVertex], In(
|     new CodeET))
|   def findDivergences(graph: QGraph[Component ++
|     Interface, _]): Unit = {
|     val t2Code = BFSTraversal[CodeGraph].over(Path1)
|     val t2Arch = BFSTraversal[ArchGraph].over(Path2)
|     graph.vertices.foreach(art => {
|       val relGraph = relGraph -- adjGraph
|       val adjGraph = new ArchGraph ++ adjGraph
|       val divGraph = divGraph -- divGraph
|       divGraph.vertices.foreach(divArt => println(
|         art("name") + "->" + divArt("name")))
|     })
```
architect. In order to maintain structural integrity, one has to check that code artifacts are interrelated if and only if their according components are related as well. A reflection model thus helps improve the software quality of an application.

Fig. 10 shows our ProGQL code for finding divergences. Line 1 defines an architecture graph using concrete vertex and edge classes. It is a directed graph that only contains outgoing edges – we need this later to describe a traversal. Line 9 defines the method findDivergences. The inclusion polymorphism exactly denotes which types of graphs are acceptable, otherwise the code will not compile. This ensures that the method only has to deal with component and interface artifacts. Without typed graphs and inclusion polymorphism, the developer would have to manually ensure the correct content of the given input set.

Starting from each architecture artifact, line 14 calculates all architecture artifacts that can be reached over architecture-to-code traceability links and code dependencies within the code model. The traversal consists of two parts. First, there is the breadth-first traversal from the starting architecture artifact to the code graph (line 11). It results in a typed code graph and gathers all code artifacts (classes, interfaces, methods, and fields) that lie on the path described in line 5. Note the elegance of the declarative, filter-based definition of the traversal. Second, there is a traversal in the opposing direction (line 12) that starts from the previously calculated code graph and searches for all related architecture artifacts. The traversal results in an architecture graph and uses the path described in line 7. In contrast to classes and interfaces, methods and fields are not directly related to architecture artifacts via an architecture-to-code traceability link. Therefore, the traversal has to follow incoming nesting links in addition to incoming traceability links. The traversal finally results in an architecture graph that may contain the root artifact that therefore has to be removed using the graph’s -= operator (line 14). Both traversals use parametric polymorphism for defining the target graph types (CodeGraph and ArchGraph). The method >>= is also parametric and results in a typed graph defined by the given typed traversal. Without parametric polymorphism, traversals were untyped and therefore could stumble over unexpected vertex and edge types. In addition to parametric polymorphism, line 14 also uses ad-hoc polymorphism as method >>= is not available to the vertex art but only to graphs. Hence, the vertex is implicitly converted at compile-time into an untyped graph. Without coercion polymorphism the code would be more cumbersome since the conversion had to be explicitly implemented.

Beside the related architecture artifacts that are reachable via the code model, the operation also has to calculate the direct adjacency (using a search depth of 1) of an architecture artifact within the architecture model. This is done in line 15. The artifact is added to a new ArchGraph and the method adj is called. Its underlying breadth-first traversal only works on vertices and (outgoing) edges that are denoted by the ArchGraph. Based on the related architecture artifacts, line 16 calculated divergences with a graph difference operator. It results in a new typed graph that has the type of the first argument and therefore is an ArchGraph. The lines that follow output the vertices of the resulting divergence graph.

We applied this ProGQL code to a real-world industrial project. The UML model has 7 components and 15 interfaces. The investigated code model consists of 8440 code artifacts (542 classes, 36 interfaces, 5469 methods, and 2393 fields in a total of 45552 lines of code). The underlying graph had 12593 vertices and 45084 edges. The ProGQL code found 47 divergences within 8.1 seconds when run on a quad-core (2.83 GHz) Windows 7 desktop PC with 8 GB RAM using an underlying Neo4J graph database. Our findings led to the decision to extend the architectural model with missing links and in addition to rework the source code to adjust it to the new architectural model.

Searching for divergences is a complex task but can be denoted in a clear and concise way. Without typed vertex and edge sets and without polymorphism the code would be more cumbersome and error prone. Furthermore, we showed that typed graphs also help define complex graph traversals in a uniform and readable way. Without ProGQL the source code would be significantly longer and therefore more difficult to understand and maintain.

4. RELATED WORK

There are many scientific and industrial approaches dealing with graphs and their implementations. [1] surveys the most important graph database models and not only focuses on structural graph representations but also mentions graph querying and traversals. Here we only cover the closest approaches and frameworks that either are based on property graphs or that deal with typed graphs.

The idea of merging graph theory and object-orientative was first published in 1992 [5], focusing on the graph’s structure instead of the data stored in graph elements.

The term property graph denotes that there are additional properties. It is mainly publicized by the TinkerPop (http://www.tinkerpop.com) community that provides a whole graph stack consisting of several graph-related frameworks and DSLs. Their Blueprints API defines interfaces for generic vertices and edges, whose types can only be identified by a type name property or a label, but there is no API support for concrete element types. ProGQL overcomes this limitation and can also deal with type inheritance. Pipes is a framework for data flow processing based on atomic units, the so-called pipes. They are lined up to pipelines that use untyped vertex and edge sets as input and output. Pipes emulate functional behavior, but since ProGQL is already a functional language, it is a better alternative for that, e.g., by providing partial functions. Our concept of a traversal is comparable to a pipeline but uses typed input and output graphs instead. In addition, every function that takes a graph as input and that produces a new typed graph as output can also be chained in this way. Gremlin is another TinkerPop framework providing an internal DSL for custom graph traversals and is based on Groovy. A Gremlin query is mapped to Pipes and therefore only works on untyped vertex and edge sets. Thus, graph polymorphism is no point for it, but it is in ProGQL. In addition, a Gremlin query has to be interpreted at runtime and is dynamically typed due to Groovy. Hence, it generally cannot perform as good as our statically typed ProGQL.

TGraph (http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEberl/MainResearch/Graphentechnologie) uses typed graph classes based on adjacency lists [3]. A TGraph is configured by a schema that is either given by a dedicated definition language, a Graph Unified
Modeling Language (GrUML) or programmatically via API. The schema exactly denotes the concrete element types in the graph and the allowed relationships between them. A generator then creates source code for concrete vertex and edge classes (see Fig. 3b) that can be supplemented by the developer. Graph queries are implemented either using an API or via the external DSL Graph Repository Query Language (GREQL). The TGgraph-approach is more extensive than ProGQL but does not provide a comparable traversal framework. Graph queries therefore have to be manually programmed. Since the GREQL runtime interpreter generates untyped sets, graph queries cannot be combined with custom graph based operations or other GREQL queries in a statically typed way. This is possible with ProGQL. Despite of its strict graph schema, TG graphs cannot provide polymorphism for graph classes. There are a few commercial or open source database systems with a property graph API. Some of them have additional schema support, an API for graph traversals, a typed query language, or a model for dealing with hypergraphs. To give concrete examples, we focus on Neo4J, sones GraphDB, and DEX.

Neo4J (http://neo4j.org) is an application-embedded graph database system that has a pure property graph API without an explicit graph class and is based on generic vertex and edge classes. There is no support for concrete types and data can only be stored using distinct primitive data types like strings, numbers, and arrays thereof. There is also an elaborated traversal API providing basic algorithms like breadth- or depth-first-traversals that can be controlled by vertex based callback handlers that are comparable to our vertex filters. But, this traversal framework is not as freely configurable as ours, e.g., the edges of the traversal path cannot be defined using arbitrary edge properties but only edge labels. To overcome such limitations, Gremlin provides an adapter to Neo4J and therefore can also be used to denote graph queries. But although vertices and edges are labeled, there are no typed vertex and edge sets. ProGQL can help with that. It also has some basic Neo4J adapters and allows for the use of typed element sets based on Neo4J’s generic elements. Together with ProGQL, the graph database then can be used with typed graphs and polymorphism.

Sones GraphDB (http://www.sones.com) overcomes the limitation of Neo4J and provides both an object-oriented graph database system that can store objects of concrete user-defined classes, and a typed graph query language with a SQL based syntax. This external DSL is embedded into the host programming language in an unostryped way. Result sets of queries are untyped and cannot be forwarded to other graph based operations without appropriate checks. It is better to use ProGQL as a query language on top of sones GraphDB. This enriches the database with typed graphs and polymorphism.

The graph database DEX (http://www.sparsity-technologies.com) is a system for high-performance graph exploration. It is based on a proprietary graph data management mechanism [6]. Although its API is not property graph based, it is related to our approach since they have a persistent DBGraph and a directly related graph pool that contains an arbitrary number of temporary RGraphs. The latter technique is analogous to our graph views. A RGraph contains references to vertices and edges of a DBGraph and since every graph in DEX has a schema denoting the types of vertices and edges, all graphs can be seen as being typed. However, the result of typed graph queries are again only untyped element sets and do not make use of the RGraph concept. RGraphs are only used for storing temporary results.

To summarize: All mentioned approaches do not provide typed element sets in a way that they can either be automatically checked at compile-time or can be used for combining graph based operations using polymorphism.

5. CONCLUSION

ProGQL is an internal DSL for custom graph traversals. The distinctive idea is the use of statically typed graphs that are defined by means of vertex and edge filters as parameters for parameterized graph classes. This kind of graph classes let the programmer fully exploit polymorphism on typed graphs. In ProGQL a custom graph based operation can exactly denote its input and output graph types. This prevents unintended use already at compile-time, avoids type-checking code, allows for overloading and thus successfully allows to improve the quality of graph processing software. On a real-world traceability project, ProGQL has proved its usability and elegance.

Acknowledgements

This research work is funded by the AIF ZIM program of the German Federal Ministry of Economics and Technology.

6. REFERENCES