Lambda Calculus Syntax's Definition and Completeness As Graph Database Querying Language

Pim van Leeuwen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
p.vanleeuwen@student.utwente.nl

ABSTRACT

Scientific interest has recently picked up graph databases: a type of database that store data in a graph-like manner. Usage of these databases rather than conventional relational databases may yield more intuitive data modelling, more intuitive querying and improved performance of query evaluation. In contrast to relational databases, however, there has not yet been set a standard for what querying language to use. Because a multitude of querying languages is used for the same goal, research considering graph database querying is slowed down, and people in this field have to learn additional languages when transferring to a new database in their field. This problem gives rise to the search for a querying language that is simple yet complete. Pokorný proposes a typed lambda calculus syntax as graph database querying language, but does not define it accurately or give an indication of its completeness. This paper aims to do that.

Keywords

Graph database, Graph querying, Lambda calculus syntax, HIT data model, Language of Terms, Functional languages, Domain specific languages, Database systems

1. INTRODUCTION

Research into graph databases has been performed ever since the mid-eighties as a cousin of the more widely used relational database. Some examples of graph database models studied are LDM, GOOD, O2, and GraphDB. Research interest had however faded away in favour of XML, semi-structured data and the semantic web. In the meantime, relational databases remained in widespread use. In the past decade, an interest in graph databases has risen again, partially due to a higher demand as result of the tremendous increase of data resulting from the rise of the Internet. As such, some of the most popular Database Management Systems (DBMS) have only recently been developed, such as Neo4J, Titan and Microsoft Azure Cosmos DB. The arguments in favour of graph databases compared to relational databases are compelling: graph databases offer intuitive conceptualisation for domains with network-like data and may even reduce computational complexity [8, 13, 20]. Applications include social networks [9], information networks [5, 14] including the semantic web [3, 15], transportation networks [4], biological networks [6, 7], program analysis [12] and criminal investigation [18].

28^{th} *Twente Student Conference on IT* February 2^{nd}, 2018, Enschede, The Netherlands.

The languages used to query from graph databases differ significantly from query languages for relational databases: not only do they allow usage of basic set operations in queries, but also for navigational ones: they approach data from a network view rather than from a table view to follow the model they represent. It is however just as vital for graph database query language to be intuitive to the users of that database since interaction with databases by their users is mainly done using queries in those languages. The effectiveness of database interactions is after all dependent on the usability of the language for the type of user for whom the database is designed. Most users choose for a declarative language to query with. Imperative querying is currently possible in other types of databases with Datatrieve [1], with Java using JReq [10] or with Odysseus using IQL [2]; there are currently no imperative querying languages for graph databases: this falls under future work (Section 4). Declarative query languages allow users to describe the requested query result without having to describe the operations the DBMS takes to get that result.

There exist some declarative graph database query languages that have added functional deatures, two of which are ProGQL [19] and Gremlin. ProGQL is a functional graph query language that is suited for statically typed graphs. Gremlin is also functional: it is path-oriented and forms query-like expressions by sequencing traversal operations and does not require static typing.

Pokorný [17] suggests applying the HIT data model's [21] idea of using typed lambda calculus syntax as a querying language to graph databases. A single lambda function is then used as a query, containing other (possibly lambda) functions inside. Doing so may open up possibilities for graph database users with a functional mindset (e.g. with experience in the lambda calculus field) and fields where queries are less path-oriented. The prospect of a complete language that is consistent with its semantics and syntax based on a few simple rules is one to look forward to- especially considering a standard for querying languages has not been set for graph databases yet. Pokorný, however, fails to demonstrate to what degree this language is complete as one for querying. This paper aims to clarify the semantics of this language and indicate its completeness with the following research questions:

- *What is the current completeness of Language of Terms (LT)?*
- *What changes to LT can be made to improve its completeness?*

The rest of the paper is structured as follows: Section 2 gives a background on graph databases and LT as proposed by Pokorný. Section 3 shows the deconstruction of the Language of Terms and how it handles the fundamental building blocks of graph querying. We discuss completeness and efficiency in Section 4 and lay out future work. Finally, Section 5 shows resulting conclusions.

2. BACKGROUND

In this section, we will first provide context for the posed problem by explaining what graph databases are, and how they use graphs to model data. We will then mention a variety of query types users may want to use and provide examples. Finally, we will provide a short background on the LT as described by Pokorný; we will continue with LT in section 3.

2.1 Graph databases

A graph database is a type of NoSQL[1] database that uses graph theory for creation and manipulation. Data about entities are stored in the nodes of a graph in the form of a set of labels and a set of key-value properties (attributes). Relationships between these objects are stored as labeled edges between those nodes. Labels depict the type of object or relationship nodes and edges have, and can be compared to a table name of a relational database or a class of an object-oriented database. For example, Figure 1 shows a graph database with node labels `Person`, `Company`, `Country` and `Branch` and with edge labels `friend`, `works_for`, `lives_in`, `located_in` and `in_branch`. Any information about an object that does not take shape as an object itself is stored as a key-value pair (e.g. `"name"` in Figure 1). This way, following the example graph, a node with label `Person` and `name "Anne"` represents a person named Anne. Similarly, an edge with label `friend` means the connected nodes represent people who are friends with each other.[2]

One should be aware that there exist some graph databases that only support edge-labeled graphs. These do not allow nodes or edges to have additional key-value properties. In such databases, such data could be stored as separate nodes and connections instead.

A graph database management system (GDBMS) is a system that interacts with such a database and allows other programs or users to request, add, modify or remove data from it using a supported query language. A GDBMS is usually inseperable from the database with which it interactions. Neo4j is an example of a GDBMS.

2.2 Query types

In graph databases, two ways of querying can be distinguished. Firstly, you can treat the graph database as a relational database and perform relational queries as you would do on a relational database (using relational algebra). That is, queries using set theoretic operations. You can request all entries (nodes) that comply with some specified first order requirement, and use set operations to combine specific results. We call these set queries. Secondly, you can use the distinctive structure of graphs to request information about the topology of data in the graph database; such queries would in SQL look like an undefined number of consecutive `JOIN`-statements. In the following two subsections, we will further elaborate on the possibilities of each type of query.

[1]Database that do not require any schema

[2]The example graph database shows use of both directed and undirected edges. Not all databases support such a distinction between edges (e.g. Neo4j).

2.2.1 *Set queries*[3]

We define a *set query* as a query that describes a request for node data or patterns of node data stored in the graph database. These queries first define sets of nodes based on invariable requirements, and then perform set operations (union, intersection and complement) including universal and existential quantifiers to finally yield the result list. Consider for example the pseudo-code set query `Give all friends of Charlie that are born after Charlie` *`or`* `work at Codus`. It describes a set of nodes with the requirements that they have a `friend`-connection with Charlie and of which the `birthday` property has a higher value than that of Charlie, and the set of nodes with the requirements that they have a `friend`-connection with Charlie and have a `works_for`-connection to a Company with the name `Codus`. Combined with the OR set operation, this yields a result list. Note that even though this is a set query that describes a request for *node data*, we may still use edges to phrase requirements for selected nodes. A simple query within the constraints of set queries is the k-hop query: a query that requests data from a fixed number (k) of hops away from some starting point. Other types of queries that fall under set queries are subgraph matching queries, clique finding queries (given some node within the clique) and connected component finding (given some node within the component). We also consider aggregation and grouping to fall under set queries.

2.2.2 *Path queries*[4]

Contrary to set queries, path queries are defined as requests for information about the topology of a graph, including reachability and finding shortest paths between nodes. For example, using Figure 1, we may want to query all pairs of people that are connected in a path consisting of `friend`-relations without specifying the length of that path. We might even query those paths as a second output. Being able to do this requires that the query language falls under the Kleene star closure property which states that arbitrarily long repetition should be possible as output given a word (in our case, query) in a starting language.

A type of query that falls under path queries is the Regular Path Query (RPQ). Such a query selects nodes of a path that belongs to a regular language over the label alphabet, including the Kleene star. RPQs are defined in Definition 1.

Definition 1. *Let L be the vocabulary of edge labels and R be a regular path expression in the form of:*
$R = l \mid R + R \mid R.R \mid R|R \mid R* \mid R? \mid (R)$ *where* $l \in L$
where $+$ denotes ordered concatenation, . denotes unordered concatenation (that is, $\forall S,T\ S.T = (S+T)|(T+S)$), $|$ denotes union, $*$ denotes Kleene star, ? denotes optional and () denotes grouping.
Then an RPQ is defined as a query that incorporates R and returns a two-columned result with nodes that are connected by R.

RPQs can be expressed as an LT query in Pokorný's definition. [17]

Most path queries can be expressed as RPQs, although it has some limitations: reverse traversal along edges and compliance to multiple RPQs is not supported (but are

[3] These category names are chosen by the author of this paper. However, the distinction itself is not new. [11]

[4]See footnote 3.

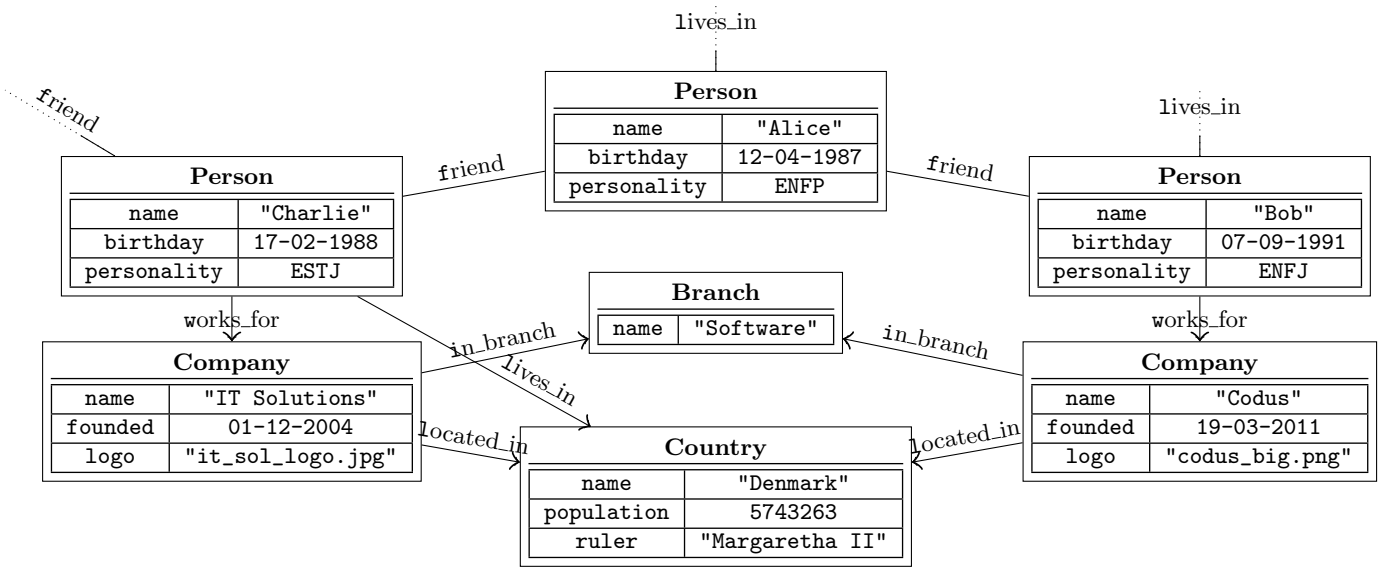

Figure 1. Example of a Graph Database that stores people, companies and countries.

supported in respectively 2RPQs[5] and CRPQs[6] or both in 2CRPQs[7]).

2.3 Language of Terms

The Language of Terms is a language defined by Pokorný as a variant of typed lambda calculus that can be used to query from (graph) databases. It is first mentioned in 1988 [16] and was recently mentioned again as possible graph query language in 2017 [17]. We will provide a formal definition in Section 3.

3. LANGUAGE OF TERMS

Pokorný defines the Language of Terms as given in *Functional Querying in Graph Databases* [17]. We would like to propose a new definition as given by Definition 3. The definition considers the scope of lambda variables and includes functions need for Kleene star and aggregation (further discussed in Section 3.2) previously unavailable. It uses types as defined in Definition 2. The set of functions defined in LT is not minimal here (some functions can be modelled with other LT functions as shown in Section 3.1). These additional functions have been added with the goal of shorter and more intuitive queries. We will not go into the types of edges or assume edges are to be returned in this paper.

Definition 2.

1. Let T_B be the set $\{BOOL, NUM, STRING\}^*$.

2. Then the set of all types $\mathbb{T}$ is defined as:

$\{(t_1 \times \cdots \times t_n) | \forall i \in \{1, \ldots, n\}\ t_i \in \mathbb{T}\} \cup$

$\{t_x \to t_y | t_x, t_y \in \mathbb{T}\} \cup T_B \cup \{node\}$

Nothing else is a type.

(We will ignore typecase when referring to types in T_B.)*

Definition 3. *Given an edge label vocabulary E, the node property keys and their respective types N and a scope of variables and their types S, we say that x is a term with type y in scope S if and only if $(x, y) \in Term(S)$. $Term(S)$ is defined as the least set such that:*

(Terms of primitive types)

$$(s, NUM) \in Term(S) \quad \text{for } s \in \mathbb{R}$$
$$(TRUE, BOOL) \in Term(S)$$
$$(FALSE, BOOL) \in Term(S)$$
$$(s, STRING) \in Term(S) \quad \text{for } s \text{ an ASCII String}$$

All real numbers, boolean values and strings are terms of the appropriate types.

(Terms of comparator function types)

$\forall A \in (T_B \backslash BOOL) \quad (=_A, (A \times A) \to BOOL)) \in Term(S)$

$\forall A, B \in (T_B \setminus BOOL) \quad (>_A, (A \times A) \to BOOL)) \forall A, B \in (T_B \setminus BOOL), \in Term(S)$

$\forall A, B \in (T_B \setminus BOOL) \quad (<_A, (A \times A) \to BOOL)) \in Term(S)$

$\forall A, B \in (T_B \setminus BOOL) \quad (\geq_A, (A \times A) \to BOOL)) \in Term(S)$

$\forall A, B \in (T_B \setminus BOOL) \quad (\leq_A, (A \times A) \to BOOL)) \in Term(S)$

Terms of base types can be compared to one another (if they are the same type) using equality, or inequality (if the base type is ordened, that is, NUM or STRING).

(Terms of logical function types)

$(\exists, (node \to BOOL) \to BOOL) \in Term(S)$

$(and, (BOOL \times BOOL) \to BOOL)) \in Term(S)$

$(or, (BOOL \times BOOL) \to BOOL)) \in Term(S)$

$(!, BOOL \to BOOL) \in Term(S)$

Basic boolean operators and, or and the inverse (!) are available in LT. The existential quantifier returns $TRUE$ if and only if there exist a node that satisfies its function argument.[8]

[5]RPQs that allow for traversal in both directions of a directed graph

[6]RPQs that are the result of conjunction of RPQs

[7]2RPQs that are also CRPQs

(Terms of arithmetic function types)
$(+, (NUM \times NUM) \to NUM) \in Term(S)$
$(-, (NUM \times NUM) \to NUM) \in Term(S)$
$(/, (NUM \times NUM) \to NUM) \in Term(S)$
$(*, (NUM \times NUM) \to NUM) \in Term(S)$

LT supports basic arithmetic functions.

(Terms of lambda function types)

$\forall (f, t_f) \in Term(S \cup \{(x_1, t_1), \ldots, (x_n, t_n)\}) \quad \forall i \in \{1, \ldots, n\}, \neg\exists y \in \mathbb{T}\ (x_i, y) \in S \quad (\lambda x_1{:}t_1, \ldots x_n{:}t_n(f), (t_1 \times \cdots \times t_n) \to t_f) \in Term(S)$

You may define a function in LT using lambda syntax. If this function is, along with its type, in $Term(\emptyset)$, its application is a query.

(Application of terms of function types)

$\forall (f, (t_1 \times \cdots \times t_n) \to s) \in Term(S) \quad \forall i \in \{1, \ldots, n\}, (x_1, t_1) \in Term(S) \quad (f(x_1, \ldots, x_n), s) \in Term(S)$

This denotes that a function of type $T \to S$ that is applied to appropriate argument(s) T are of type S.

(Terms of tuple types)

$\forall \{(x_1, t_1), \ldots, (x_n, t_n)\} \subseteq Term(S) \quad ((x_1, \ldots, x_n), (t_1 \times \cdots \times t_n)) \in Term(S)$

You may 'embed' multiple terms into a single tuple term.

(Tuple extraction)

$\forall ((x_1, \ldots, x_n), (t_1 \times \cdots \times t_n)) \in Term(S) \quad \forall i \in \{1, \ldots, n\}$
$(x[i-1], t_i) \in Term(S)$

Using bracket notation, items may be extracted from tuple terms.

(Functions based on edge connections)

$\forall e \in E \quad (e, (node, node) \to BOOL) \in Term(S)$

Edge labels may be used as functions to test whether nodes are connected by such an edge.

(Retrieval of node properties)

$\forall (p, t) \in N \quad \forall (n, node) \in Term(S) \quad (n.p, t) \in Term(S)$

Node properties may be retrieved from terms of type node via dot notation.

(Terms of other functional types)

$\forall A, B \in \mathbb{T} \quad (fold_{A,B}, (B \times A \to B) \times B \times (A \to BOOL) \to B)) \in Term(S)$ *where the term of type $(A \to BOOL)$ is a query.*

Fold is a function used for aggregation of a result set; application of fold always results in a query. Its third argument is the query to be aggregated and its second argument is some value `res`*. The first argument is a function that is repeatedly performed on the* `res` *value and each elements from the query.* `res` *is then returned.*

$\forall A, B \in \mathbb{T} \quad (foldgroup_{A,B,C}, (((B \times A) \to B) \times B \times (A \to BOOL) \times (B \to C)) \to (B \to BOOL))) \in Term(S)$
where the term of type $(A \to BOOL)$ is a query.

Foldgroup performs the same operations as fold but yields a collection of values rather than a single one. Its fourth argument is performed on each element from the query result set, and entries with the same result are grouped together in the foldgroup result set.

$\forall (f, (node \times node) \to BOOL) \in Term(S) \quad (repeat, ((node \times node) \to BOOL) \to ((node \times node) \to BOOL)) \in Term(S)$

Repeat is a higher level function that transforms a function that selects two nodes based on some relationship to a function that selects two nodes that are Kleene star connected via that relationship.

A query q is a term of type $(T_1 \times \cdots \times T_n) \to BOOL$ for which $(q, (T_1 \times \cdots \times T_n) \to BOOL) \in Term(\emptyset)$

The Language of Terms is defined as the set of all queries.

3.1 Semantics

We will now cover the semantics of LT by giving the meaning of valid terms according to Definition 3.

(constants)

$c, (c, NUM) \in Term(S)$	The numeric value of c.
$c, (c, BOOL) \in Term(S)$	The true/false value of c.
$c, (c, STRING) \in Term(S)$	The string value of c.

(functions)

$fold$	(Given in Section 3.2.2)
$foldgroup$	(Given in Section 3.2.2)
$=_A$	Function that returns a BOOL value indicating whether the given terms have equal types and values.
and	Function that returns TRUE if both its arguments are TRUE.
!	!FALSE = TRUE / !TRUE = FALSE
$or(A.B)$	$!(and(!(A), !(B)))$
$>_{NUM}$	Function that returns a BOOL value indicating whether the first term has a greater value than the second term.
$>_{STRING}$	Function that returns a BOOL value indicating whether the the value of the first term is alphabetically before the value of the second term.
$<_{NUM}\ (A, B)$	$>_{NUM}\ (B, A)$
$<_{STRING}\ (A, B)$	$>_{STRING}\ (B, A)$
$<_{STRING}\ (A, B)$	$>_{STRING}\ (B, A)$
$\geq_{NUM}\ (A, B)$	$or(>_{NUM}\ (A, B), =_{NUM}\ (A, B))$
$\geq_{STRING}\ (A, B)$	$or(>_{STRING}\ (A, B), =_{STRING}\ (A, B))$
$\leq_{NUM}\ (A, B)$	$or(>_{NUM}\ (B, A), =_{NUM}\ (A, B))$
$\leq_{STRING}\ (A, B)$	$or(>_{STRING}\ (b, a), =_{STRING}\ (A, B))$
$+, -, *, /$	(Same meaning as in arithmetic syntax)

$\lambda x_1{:}t_1, \ldots, x_n{:}t_n(f(\ldots))$	Function that accepts n arguments of types t_1, t_n and returns the result of $f(\ldots)$.
$repeat(F)(A, B)$	$or(f(A, B), \exists(\lambda x{:}node(TRUE), \lambda x{:}node(and(F(A, x), repeat(F)(x, B))))$
$\exists$	This function takes a function and returns wether there exists a node input such that it returns $TRUE$.

3.2 Insights

In this section we will consider a variety of insights risen by examination of LT that are worthy of discussion.

3.2.1 Types of lambda variables

While the Language of Terms is based on typed lambda calculus, it was originally not defined what the type of a query should be when the language is used to query. Rather, a query would always be in the form of a lambda function application without indication of the types of the lambda variables. For example, the current version of LT uses $\lambda x, y(f(x, y))$ rather than $\lambda x{:}node, y{:}num(f(x, y))$. Since each function used in lambda queries has a static type (typically a function type that yields a BOOL), it would be a good idea to also statically type the lambda variables on which these functions are performed.

In most querying cases, a user wishes for a selection of nodes to be returned. *It is however possible that elements from another type (with functions that use aggregation) are requested.* Lambda functions that are not queries but embedded in one may also need to use lambda variables of different types. It is therefore not an option to always assume the type of lambda variables to be node. We herewith suggest that lambdas should always be statically typed by the user. This yields a language in which queries may be longer, but any mismatch of user intention and query functionality is found quickly and in which exist no ambiguity. We will therefore not write:

λ *x,y,z(. . .)*

We will instead use static typing as so:

λ *x:node,y:node,z:int(. . .)*

This second notation is used throughout this paper.

3.2.2 Aggregation using fold

Aggregation and aggregation with grouping cannot be done in the original LT. We therefore propose to add the following two functions to the original definition of LT:

$(fold_{A,B}, (((B \times A) \to B) \times B \times (A \to BOOL)) \to B)) \in Term(S)$

$(foldgroup_{A,B,C}, (((B \to BOOL) \times A) \to (B \to BOOL)) \times (B \to BOOL) \times (A \to BOOL) \times (B \to C)) \to (B \to BOOL)) \in Term(S)$

for any $A, B, C \in \mathbb{T}$ where $(A \to BOOL)$ is a query.

Fold is a function used for aggregation without grouping. The definition of fold is: $fold_{A,B}(F, X, Q) =$

Let $u = \exists(c, A) \in Term(S), Q(c) = TRUE$

for u and some p where $(p, A) \in Term(S)$ and $Q(p) = TRUE$ let $fold_{A,B}(F, X, Q) = fold_{A,B}(F, F(X, p), \lambda x{:}A($ $and(Q(x), ! = (x, p))$.

otherwise $fold_{A,B}(F, X, Q) = X$

Careful observers may notice that if F is not commutative, the result of $fold_{A,B}$ depends on which p is picked. This is also the case for $foldgroup_{A,B,C}$. We therefore always advise commutativity for these functions to avoid ambiguous results.

let $u' = \exists(c, A) \in Term(S), Q(c) = TRUE \wedge \neg\exists c', X(c') \wedge S(c') = S(c)$ and let $u = \neg u' \wedge \exists(c, A) \in Term(S), Q(c) = TRUE$ and

for u' let $foldgroup_{A,B,C}(F, X, Q, S) = foldgroup_{A,B,C}(F, \lambda x(or(X(x), = (x, P))), \lambda x(and(Q(x), ! = (x, P)), S), (P, A) \in Term(S), Q(c) = TRUE \wedge \neg\exists c', X(c') \wedge S(c') = S(c)$

for u let $foldgroup_{A,B,C}(F, X, Q, S) = foldgroup_{A,B,C}(F, \lambda x(or(and(X(x), ! = (x, c'), x = F(c', P))), \lambda x(and(Q(x), ! = (x, P)), S), (P, A) \in Term(S), Q(c) = TRUE \wedge S(c') = S(P)$

otherwise $foldgroup_{A,B,C}(F, X, Q, S) = X$

An informal description of the functionality of these functions can be found in Definition 3.

3.3 Deconstruction

In this section, we will give a set of building blocks (read: capabilities) that are necessary for specific queries and determine whether these building blocks are in LT.

3.3.1 Node access

One of the most trivial building blocks is access to the graph by the query language; it is essential for graph database query languages. LT supports graph access. See Translation 1.

Translation 1 (Node access in LT)**.**
Let $q = \lambda x_1{:}node, x_2{:}t_1, \ldots, x_n{:}t_n(F(x_1, \ldots, x_n))$
This LT query is defined as the query of which the result is the cross product of all lambda variable possiblities for which F yields true. If the user defines F to be TRUE for some $x_1, \ldots, x_n$, then the result set's first column will contain nodes. This way, LT supports node access.

3.3.2 Edge access

Besides nodes, the database graph also stores relationships. These can be accessed by LT to select nodes as a different form of selection. See Translation 2.

Translation 2 (Edge access in LT)**.**
Let e be some edge label (so $e \in E$).
Then $(e, (node \times node) \to BOOL) \in Term(S)$ can be used to test whether two nodes are connected by an edge with that label.

3.3.3 Set difference

Set difference can usually be expressed as a combination of set union and negation. However, in querying, negation is a form of set difference (e.g. $\neg s = G \setminus s$). Set difference cannot be modeled with other found building blocks. Translation 3 shows how to use set difference in LT.

Translation 3 (Set difference in LT)**.**

Let q_1 and q_2 be queries with type $(T_1 \times \cdots \times T_n) \to BOOL$.

Then $q_1 \setminus q_2$ can be queried with:

$\lambda x_1{:}T_1, \ldots, x_n{:}T_n(q_1(x_1, \ldots, x_n)$ *and* $!q_2(x_1, \ldots, x_n))$

3.3.4 Set carthesian product

In the context of graph querying, a carthesian product is modeled by returning a result list with the columns of each part added. A set containing two elements can be

returned in a single query result entry. Ccontrary to set union, difference and intersection, the carthesian product non-strictly increases the number of columns in a result set. Since no other buidling block can do that, it cannot be modeled by them. Translation can be found in Translation 4.

Translation 4 (Carthesian product in LT)**.**
Let q_1 and q_2 be queries with respective types $(T_1 \times \cdots \times T_n) \to BOOL$ and $(T'_1 \times \cdots \times T'_m) \to BOOL$.

Then the carthesian product of q_1 and q_2 is:

$\lambda x_1{:}T_1, \ldots, x_n{:}T_n, y_1{:}T'_1, \ldots, y_m{:}T'_m(and(q_1(x_1, \ldots, x_n), q_2(y_1, \ldots, y_m))$

3.3.5 Selection

In relational algebra and here, selection denotes selection of a subset from a query result where an (in)equality holds for one of the columns of the result list entries. Reducing the size of a query result is not only an effect of selection, but also of aggregration (with grouping and arithmetic). Selection can however not be modeled with aggregation. Translation 5 shows how to use selection in LT.

Translation 5 (Selection in LT)**.**
Let q be some query with arguments $x_1, \ldots, x_n$ of types $t_1, \ldots, t_n$ respectively. Then a selection with $x_i > c$ can be written as:

$\lambda x_1{:}t_1, \ldots, x_n{:}t_n(and(q(x_1, \ldots, x_n), > (x_i, c)))$

Similarly other inequalities and equalities can be modeled in LT.

3.3.6 Arithmetic

Arithmetic can be seen as an extension of aggregation and selection that introduces functions to manipulate numbers. No other building block allows for number manipulation. Translation follows from Definition 3.

3.3.7 Existential quantifier

The existential quantifier $\exists$ is a method of testing whether a function results in true for a single element in a set. In the context of graph databases, this means its type is $((A \to BOOL) \times (A \to BOOL)) \to BOOL$. The function can be interpreted as: *Return whether there is an element from the set of elements for which the first argument returns true.* The existential quantifier cannot be modeled using other building blocks. Translation is elementary since $\exists$ is a function in LT.

3.3.8 Aggregation

Aggregation is an operation that reduces the size of the result set of a query, like selection; it can however not be modeled by selection. Since no other building block reduces the result set size of a query, we consider aggregation to be a fundamental building block. Translation 6 shows how to use aggregation in LT.

Translation 6 (Aggregation in LT)**.**
Let us consider an example query $q = \lambda x_2{:}node, x_1{:}int(f)$. Let us say the user wants to have the sum of all entries in the second column of the result of this query. This is given by:

$q' = fold_{(node \times INT), BOOL}(\lambda x{:}int, y{:}(node \times int)(x + y[1]), 0, \lambda x_2{:}node, x_1{:}int(f))$

If the user wants to count the number of elements in q intead, this is given by:

$q' = fold_{(node \times INT), BOOL}(\lambda x{:}int, y{:}(node \times int)(x + 1), 0, \lambda x_2{:}node, x_1{:}int(f))$

Similarly, any other type of aggregation can be done my changing the aggregating function or the starting value.

3.3.9 Grouping

Grouping is an extension of aggregation that allows aggregation over specific sets of rows from the query result list. Aggregation with grouping cannot be modeled using using other building blocks (except in multiple consecutive queries, which we do not consider valid). Translation 7 shows how to use grouping in LT.

Translation 7 (Grouping in LT)**.**
The definition of the LT function used for grouping ($foldgroup_{A,B,C}$) is given in Section 3.2.2. We can use this for example to sum the salaries of each department of a company for a database containing such data. The query would look like this:

> $foldgroup(\lambda x{:}num, y{:}num(x + y), 0, \lambda\ hidden\ p{:}node, d{:}string, s{:}num(and (=_{STRING}\ (p.department, d), =_{STRING} (p.salary, s))), \lambda x{:}string, s{:}string(x))$

The aggregating function, starting value or grouping indentifier can all be adapted to suit the needs of the aggregation.

3.3.10 Kleene star for edges

One vital part of RPQs is the inclusion of the Kleene star for path queries. As extension of edge access, it allows querying for nodes connected by an undefined number of consecutive edges of the same type. Kleene star for edges can only be modeled by other building blocks in two consective queries, which we do not count. LT allows querying using the Kleene star with the *repeat* function.

3.3.11 Grouped Kleene star for edges

Grouped Kleene star is an extension of Kleene star that allows it to cover a concatenated list of edge labels rather than a single edge label. If we take a look at the definition of RPQs (Definition 1), we can see that grouped Kleene star for edges, edge access and set difference are enough to evaluate RPQs and even CRPQs. LT allows querying using grouped Kleene star with the *repeat* function.

We now consider some building blocks that follow from this base set.

3.3.12 Set union

Set union is an operation that delivers the union result of two queries of which the results are of the same type. Set union can be derived from set difference. Let q_1 and q_2 be queries with result sets s_1 and s_2 respectively, of the same type t that has universe U_t. Then $s_1 \cup s_2 = U_t \setminus (U_t \setminus s_1 \setminus s_2)$.

3.3.13 Set intersection

Set intersection has similarities with set difference, but is fundamentally different in that it is symmetrical. As set operation, it can be defined using set difference: $q_1 \cap q_2 = G \setminus (G \setminus (q_1 \setminus q_2) \setminus (q_2 \setminus q_1))$.

3.3.14 Projection

In relational algebra, projection is defined as returning all query result entries but with a subset of all available columns, defined by the user. Since the result of a query is a set and removing columns may result in duplicate entries, use of projection may not only result in fewer columns in the result set but also in fewer entries. Projec-

tion can be modeled using the existential quantifier function $\exists$ or the $foldgroup$ function.

3.3.15 Universal quantifier

For universal quantifiers holds that $\forall(f_1, f_2) = !\exists(f_1, !f_2)$. Therefore, the universal quantifier can be modeled using set difference and the existential quantifier.

4. DISCUSSION AND FUTURE WORK

As we have shown, LT supports the evaluation of RPQs. There are however variants of RPQs that have not yet been tested for compatability with LT, such as NREs and RQMs [11]. The inclusion of such queries could result in much higher completeness of the language with only a minor inclusion of syntax.

We have shown that LT supports specific building blocks, but not how much computational power the evaluation of queries implementing these building blocks would take. This requirement could potentially be too high for practical use of the query language and require changes to allow the query to be expressed such that its complexity is decreased.

One function that would qualify as building block but is not considered in this paper is the power set function of type $((A \to BOOL) \to ((A \to BOOL) \to BOOL))$. Perhaps some building blocks can be modelled using such a function. This is however for future work.

In the background section, we mentioned the existence of imperative query languages and that no such languages exist for graph databases. These languages may potentially yield high completeness or usability and will have to be researched.

A feature often used in other databases is ordering results. LT could implement an extension of queries that allow the user to order the results; that, however, goes against the notion that the result of a database query is a set. Future work may include a transition from result sets to ordened lists and the introduction of ordening functions.

5. CONCLUSION

Graph databases have been shown to improve the performance of data retrieval, and are nowadays popular in research. There exist querying languages for them, but a standard has not been formed yet (as SQL has for relational databases). Pokorný proposed Language of Terms as a graph querying language in *Functional Querying in Graph Databases* [17]. It is a querying language based on lambda calculus which shows potential for efficient querying with few syntax rules but lacked a formal definition or an indication of its completeness in its state. To solve this, we have gathered a set of mutually exclusive query language capabilities that indicate the completeness of a graph database query language and proposed a new version of LT that has those capabilities. A summarized result is given in Figure 2. Capabilities marked with an asterisk (*) are not within LT in its current state but would be supported if the mentioned functions were added. With this, we give a critique of the LT and suggest changes to it such that it has the proposed capabilities.

Building block	LT support
Node access	Yes
Edge access	Yes
Set difference	Yes
Set carthesian product	Yes
Arithmetic	Yes
Existential quantifier	Yes
Universal quantifier	*follows from other blocks*
Aggregation	With addition of $fold^*$
Grouping	With addition of $foldgroup^*$
Kleene star	With addition of $repeat^*$
Grouped Kleene star	With addition of $repeat^*$
Set union	*follows from other blocks*
Set intersection	*follows from other blocks*
Projection	*follows from other blocks*
Ordering	No
RPQs	*follows from other blocks*
CRPQs	*follows from other blocks*
2RPQs	No
C2RPQs	No
NREs, RQMs, ...	*Unknown*

Figure 2. Summary of the found results

6. REFERENCES

[1] *VAX DATATRIEVE Reference Manual*, May 1992.

[2] Imperative query language (IQL). by Universität Oldenburg, Dec 2015.

[3] M. Arenas, C. Gutierrez, and J. Pérez. An extension of SPARQL for RDFS. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5005 LNCS:1–20, 2008.

[4] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[5] P. de Solla and D. J. Networks of scientific papers. *Science*, 149(3683):510–515, 1965.

[6] Y. Deville, D. Gilbert, J. van Helden, and S. Wodak. An overview of data models for the analysis of biochemical pathways. *Briefings in bioinformatics*, 4(3):246–259, 2003.

[7] M. Graves, E. R. Bergeman, and C. B. Lawrence. Graph database systems. *IEEE Engineering in Medicine and Biology Magazine*, 14(6):737–745, Nov 1995.

[8] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[9] R. A. Hanneman and M. Riddle. *Introduction to social network methods*. University of California, Riverside, Riverside, CA, 2005.

[10] M.-Y. Iu, E. Cecchet, and W. Zwaenepoel. JReq: Database queries in imperative languages. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6011 LNCS:84–103, 2010.

[11] L. Libkin, W. Martens, and D. Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, Mar. 2016.

[12] Y. Liu and S. Stoller. Querying complex graphs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3819 LNCS:199–214, 2005.

[13] M. Mainguenaud. Modelling the network component of geographical information systems. *International Journal of Geographical Information Systems*, 9(6):575–593, 1995.

[14] W. Nejdl, W. Siberski, and M. Sintek. Design issues and challenges for RDF- and schema-based peer-to-peer systems. *SIGMOD Rec.*, 32(3):41–46, Sept. 2003.

[15] J. Pérez, M. Arenas, and C. Gutierrez. NSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010.

[16] J. Pokorný. A function: Unifying mechanism for entity-oriented database models. In *Proceedings of the Seventh International Conference on Enity-Relationship Approach: A Bridge to the User*, pages 165–181, Amsterdam, The Netherlands, The Netherlands, 1989. North-Holland Publishing Co.

[17] J. Pokorný. Functional querying in graph databases. In *Intelligent Information and Database Systems*, pages 291–301, Cham, 2017. Springer International Publishing.

[18] A. Sheth, B. Aleman-Meza, I. Arpinar, C. Bertram, Y. Warke, C. Ramakrishanan, C. Halaschek, K. Anyanwu, D. Avant, F. Arpinar, and K. Kochut. Semantic association identification and knowledge discovery for national security applications. *Journal of Database Management*, 16(1):33–53, 2005.

[19] N. Tausch, M. Philippsen, and J. Adersberger. A statically typed query language for property graphs. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, pages 219–225, New York, NY, USA, 2011. ACM.

[20] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[21] J. Zlatuska. Hit data model data bases from the functional point of view. In *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden.*, pages 470–477, January 1985.