On the Semantics of Cypher's Implicit Group-by*

Filip Murlak
University of Warsaw
Poland
fmurlak@mimuw.edu.pl

Jan Posiadała
Nodes and Edges
Poland
jan.posiadala@gmail.com

Paweł Susicki
Nodes and Edges
Poland
pawel.susicki@gmail.com

Abstract

Cypher is a popular declarative query language for property graphs. Despite having been adopted by several graph database vendors, it lacks a comprehensive semantics other than the reference implementation. This paper stems from Cypher.PL, a project aimed at creating an executable (and readable) semantics of Cypher in Prolog, and focuses on Cypher's implicit group-by feature. Rather than being explicitly specified in the query, in Cypher the grouping key is derived from the return expressions. We show how this becomes problematic when a single return expression mixes unaggregated property references and aggregating functions, and discuss ways of giving this construct a proper semantics without defying common sense.

CCS Concepts • **Information systems → Query languages**.

Keywords Cypher query language, implicit group-by

ACM Reference Format:
Filip Murlak, Jan Posiadała, and Paweł Susicki. 2019. On the Semantics of Cypher's Implicit Group-by. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages (DBPL '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3315507.3330196

1 Introduction

Cypher [16, 20] is a declarative query language for the property graph data model, inspired by SQL, with the concept of pattern matching taken from SPARQL. Being a declarative language, Cypher focuses on what to retrieve from a graph, not how to retrieve it. Cypher was originally created by Neo Technology for its graph database Neo4j, but was opened up through the openCypher initiative [17] in October 2015 and has since been adopted by several graph database vendors. OpenCypher is a community of vendors, researchers, and users, whose goal is to help Cypher become the standard query language for property graphs. This is to be achieved by delivering a number of artifacts, including a reference documentation, a grammar specification, the Technology Compatibility Kit (an extensive collection of test scenarios), and a language specification. This last subgoal has not been reached yet, but a formal semantics of a limited fragment has been made available recently [4, 5]. It covers the core of the language, but omits many of its more advanced features.

Cypher.PL is a project aiming at creating an executable and readable semantics of Cypher in Prolog, in the spirit of [6–8, 11]. Currently, it supports the language to the extent sufficient to pass all the test scenarios from the Technology Compatibility Kit [18], which makes its scope much wider than that of [4, 5]. At the moment, Cypher.PL is the most accurate semantics of Cypher. Designing a clean general semantics for the advanced features of the language, fully compatible with the test scenarios and the reference documentation, is not always easy. One of such problematic features is Cypher's implicit group-by construction.

Grouping operations are often more complicated than it may seem. Even SQL, despite its explicit `GROUP BY` clause, displays some unexpected behaviour. For instance, in most popular implementations, like Oracle (12c Release 1), MySQL (v8.0), PostgreSQL (v10.0), SQL Server (2014 Express Edition), the first two of the following queries work as intended, but the last one, rather unexpectedly, gives the error "`Not a GROUP BY expression`" (or similar).

```
SELECT a + 1 FROM t GROUP BY a;
SELECT a + 1 FROM t GROUP BY a + 1;
SELECT a     FROM t GROUP BY a + 1;
```

A notable exception is SQLite (v3.26), which treats the last query just like the first two. The SQL standard [10] avoids all difficulties by allowing only a sequence of column names in the `GROUP BY` clause, thus excluding the last two queries.

In Cypher, the situation is much more complex, because the grouping key is not explicitly specified in the query, but derived from the return expressions. For instance, in the query below, it is supposed that the user expects publications to be grouped by author's name and the total impact factor of each author is to be returned.

```
match (author:Author) --> (article:Article),
      (article:Article) --> (journal:Journal)
return author.name, sum(journal.IF)
```

*Supported by Poland's National Science Centre grant 2018/30/E/ST6/00042.

DBPL '19, June 23, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6718-9/19/06…$15.00
https://doi.org/10.1145/3315507.3330196

The implicit group-by feature is excluded from the fragment covered by the partial semantics in [4], and alternative proposals for graph query languages choose the standard approach with an explicit grouping key [1]. These decisions become understandable in the light of our further discussion. Nevertheless, being a distinctive feature of Cypher and enjoying significant support among developers, implicit group-by deserves a closer look; even more so in the context of the recent push for a single property graph query language [9]. In this paper we try to understand the weaknesses of implicit group-by and investigate ways of fixing it without compromising the intended use-cases.

We begin by showing how implicit group-by becomes problematic when a single return expression mixes unaggregated property references and aggregating functions (Section 2): we use a variant of examples initially communicated at the First oCIM in February 2017 [15]. Next, we investigate ways of giving this construct a general semantics: we collect a list of postulates a reasonable semantics should satisfy (Section 3), we appraise the implemented semantics for unmixed expressions and argue why an equally canonical general semantics does not exist (Section 4), we propose three candidate semantics of decreasing accuracy but increasing simplicity (Sections 5–7), and discuss their computability and feasibility (Section 8). Finally, we venture to draw some conclusions for the future of implicit group-by (Section 9).

2 Heuristics

The semantics of the implicit group-by, as implemented in Neo4j, is neither clear nor clean. We illustrate the problems with a handfull of examples. We begin with a query that does not involve grouping.

Example 2.1. To avoid clauses specific to the property graph model, we use the UNWIND clause to manually build input for the group-by operation.

```
unwind [{a:1, b:2}, {a:2, b:1}] as x
return x.a + x.b
```

The UNWIND clause interprets the given list of maps as a sequence of *environments*, analogous to rows of a table, or tuples in the named variant of relational algebra. Each environment binds identifiers a, b to values. These environments are then ranged over by the variable x. For each environment x the value of the expression in the RETURN clause is output as a row of the answer. The expression $x.a$ is an *identifier reference* and it evaluates to the value to which the environment x binds the identifier a.

```
+-----------+
| x.a + x.b |
+-----------+
| 3         |
| 3         |
+-----------+
```

In general, the RETURN clause contains a comma-separated list of expressions built from identifier references, numeric values, mathematical operators

+, -, *, /, %, ^,

and a rich set of mathematical functions available in Cypher, including basic functions like

abs(), sqrt(), log(), exp(), sin(), cos(), tan(),

as well as mathematical constants

e(), pi().

Implicit group-by is invoked whenever the RETURN clause contains a call of an aggregating function. Cypher supports the well-known aggregating functions

avg(), count(), max(), min(), sum(),

slightly more advanced statistical functions

percentileCont(), percentileDisc(), stdev(), stdevP(),

and the `collect()` function, which returns the list of all values in the group. We exclude the latter from our considerations and limit ourselves to numeric values. Each aggregating function takes as input an arbitrary expression, as described above. The functions `percentileCont()` and `percentileDisc()` take an additional numeric parameter. The usual `count(*)` is also available. Aggregating functions can be used freely in more complex expressions, except that they cannot be nested.

In the explicit group-by operation, known from classical query languages like SQL, the user explicitly indicates the *grouping key*, a list of expressions to be used when grouping. The input set of environments (tuples) is split into groups according to the values of these expressions, and aggregating functions are evaluated over the resulting groups. In Cypher, the grouping key is derived automatically from the expressions used in the RETURN clause.

Example 2.2. Let us modify the query from Example 2.1 by including the expression `count(*)` in the RETURN clause.

```
unwind [{a:1, b:2}, {a:2, b:1}] as x
return x.a + x.b, count(*)
```

The environments get grouped by the value of the expression `x.a + x.b`, and for each group the value of the expression `count(*)` is computed.

```
+----------------------+
| x.a + x.b | count(*) |
+----------------------+
| 3         | 2        |
+----------------------+
```

Example 2.3. The grouping key may consist of more than one expression.

```
unwind [{a:1, b:2}, {a:2, b:1}] as x
return x.a, x.b, count(*)
```

Now, environments get grouped by the value of the pair of expressions `x.a`, `x.b`.

```
+----------------------+
| x.a | x.b | count(*) |
+----------------------+
| 1   | 2   | 1        |
| 2   | 1   | 1        |
+----------------------+
```

Let us speculate on the intended semantics, based on these examples. Apart from numeric values, standard mathematical functions, and operations, there are two kinds of primitives in the expressions: identifier references like `x.a` and aggregating function calls like `count(*)`. Expressions involving unaggregated identifiers are evaluated in each environment separately, and should have the same value for all environments constituting a group. Expressions involving aggregating functions are evaluated in a group as a whole. They make no sense for a single environment. In particular, one cannot meaningfully group by the value of such an expression. Thus, as a grouping key one should take those expressions in the RETURN clause that contain unaggregated identifier references, and expressions containing aggregating functions should be evaluated over the resulting groups. This simple and natural semantics accurately describes what is happening—as long as all expressions listed in the RETURN clause can be split into those containing unaggregated identifier references and those containing aggregating functions. This, however, is not guaranteed by Cypher's syntax.

Example 2.4. Cypher allows arbitrary mixing of identifiers and aggregating functions.

```
unwind [{a:1, b:2}, {a:2, b:1}] as x
return x.a + x.b + count(*)
```

This immediately makes things difficult. Some grouping is necessary, because an aggregating function is used, but how do we choose the grouping key?

The necessary condition is that for all environments within a group, the value of the return expression is the same. This is guaranteed when all environments within a group agree on identifiers used in the return expression. But is this the coarsest partition into groups that guarantees this condition? For this specific query, it suffices to group by the value of the expression `x.a + x.b`. And indeed, this is what the reference implementation does.

```
+----------------------+
| x.a + x.b + count(*) |
+----------------------+
| 5                    |
+----------------------+
```

Choosing the grouping key based on the content of the RETURN clause is not obvious. The next example shows that the reference implementation is not doing a great job here.

Example 2.5. Let us change the order of the summands in the RETURN clause of the query from Example 2.4.

```
unwind [{a:1, b:2}, {a:2, b:1}] as x
return x.a + count(*) + x.b
```

Rather unexpectedly, the query now returns two rows.

```
+----------------------+
| x.a + count(*) + x.b |
+----------------------+
| 4                    |
| 4                    |
+----------------------+
```

It appears that the grouping key is now the pair `x.a`, `x.b`, rather than the single expression `x.a + x.b`. The reason is most likely that the two summands got separated by the call to an aggregating function (the reference implementation does not disclose its policy of choosing the grouping key).

The last two examples show that the current treatment of mixed expressions in the reference implementation does not preserve commutativity of addition. This is clearly undesirable. But can it be fixed at all? In the following sections we will see that it is problematic.

3 What Is a Semantics for Group-by?

We shall now abstract from the specific functions, aggregating functions, and domains available in Cypher, and consider an abstract implicit group-by operation, independent not only from the query language, but also from the data model.

For simplicity, we assume a single domain $\mathbb{D}$. We fix a finite set Fun of function symbols and a finite set Agg of aggregating function symbols. Each function symbol $f \in \mathsf{Fun}$ has its arity $\mathrm{ar}(f)$ and interpretation $\hat{f} : \mathbb{D}^r \to \mathbb{D}$ where $r = \mathrm{ar}(f)$. Each aggregating function symbol $F \in \mathrm{Agg}$ is unary and is interpreted as a function $\hat{F}$ mapping finite multisets of elements of $\mathbb{D}$ to single elements of $\mathbb{D}$.

While Cypher queries run over graph data, the group-by operation is always applied to the so-called driving table, which is a set of environments that evolves as the successive clauses of the query are evaluated. Consequently, we can assume that the abstract implicit group-by takes as input

- a finite set V of environments binding identifiers to values from $\mathbb{D}$, and
- a sequence of return expressions $e_1, e_2, \ldots, e_k$ (those listed in the RETURN clause of the query),

and is supposed to return a multiset of tuples. We assume that all environments in V bind the same identifiers. That is, there is a finite set Ide such that each $\eta \in V$ is a function $\eta : \mathsf{Ide} \to \mathbb{D}$. We write $\mathbb{D}^{\mathsf{Ide}}$ for the set of all functions from Ide to $\mathbb{D}$. Each expression e_i is a term over $\mathsf{Fun} \cup \mathrm{Agg} \cup \mathsf{Ide}$, where elements of Ide are treated as nullary function symbols. We keep the assumption that symbols from Agg are not nested. Occurrences of symbols from Ide inside subterms of the form $F(e')$ with $F \in \mathrm{Agg}$ are called *aggregated*; the remaining occurrences are called *unaggregated.* An expression is *mixed* if it contains a symbol from Agg and an unaggregated occurrence of a symbol from Ide.

Let us note that by letting V be a set rather than a multiset we do not lose generality. We can simulate repetitions in V

by introducing auxiliary identifiers, not used in the return expressions, that distinguish copies of the same environment.

The value $[\![e]\!]^{\eta}_{U}$ of expression e with respect to an environment η and a finite set U of environments (a group) is defined by induction on the structure of e. For $a \in \mathsf{Ide}$, $f \in \mathsf{Fun}$, and $F \in \mathsf{Agg}$, we have

$$[\![a]\!]^{\eta}_{U} = \eta(a)\,,$$
$$[\![f(e_1, e_2, \ldots, e_p)]\!]^{\eta}_{U} = \hat{f}\left([\![e_1]\!]^{\eta}_{U}, [\![e_2]\!]^{\eta}_{U}, \ldots, [\![e_p]\!]^{\eta}_{U}\right),$$
$$[\![F(e)]\!]^{\eta}_{U} = \hat{F}\left(\left\langle [\![e]\!]^{\theta}_{U} \,\middle|\, \theta \in U \right\rangle\right),$$

where by $\langle o_i \mid i \in I \rangle$ we mean the multiset of objects o_i with i ranging over the set I. If e is a term over $\mathsf{Fun} \cup \mathsf{Ide}$, then $[\![e]\!]^{\eta}_{U}$ depends only on η, and we refer to the common value as $[\![e]\!]^{\eta}$. We say that U is *consistent* with e if for all $\eta, \eta' \in U$ it holds that $[\![e]\!]^{\eta}_{U} = [\![e]\!]^{\eta'}_{U}$. We write $[\![e]\!]_{U}$ for the common value. Note, that for $\eta'' \notin U$, it is possible that $[\![e]\!]^{\eta''}_{U} \neq [\![e]\!]_{U}$.

We restrict our attention to semantics that are based on actual grouping. That is, when an implicit group-by is performed for V and $e_1, e_2, \ldots, e_k$, the semantics chooses a partition of V into groups $V_1, V_2, \ldots, V_n$ for some n and returns the values of expressions $e_1, e_2, \ldots, e_k$ determined for each group separately. The chosen partition must be consistent with the return expressions: each group V_i must be consistent with each return expression e_j. The answer is the multiset of tuples $\left([\![e_1]\!]_{V_i}, [\![e_2]\!]_{V_i}, \ldots, [\![e_k]\!]_{V_i}\right)$ for $i = 1, 2, \ldots, n$.

Let us digress a little and see how consistent partitions help understand the behviour of SQL implementations signalled in the introduction. In SQL we group according to the values of expressions specified in the `GROUP BY` clause. Thus we obtain a partition of tuples into groups. For each group we compute the value of each expression e from the `SELECT` clause. The values of e are well defined iff the partition is consistent with e. Why, then, not all implementations return the answers one expects for the three queries from the introduction? Testing consistency at run time is easy. What is challenging is ensuring consistency over any input data. This requires deciding if the values of expressions from the `GROUP BY` clause functionally determine the values of all expressions from the `SELECT` clause. SQLite does not guarantee anything and uses the value yielded by an arbitrary tuple from the group. The remaining implementations test a heuristic sufficient condition and complain if it fails; this leads to false negatives, as in our examples. The approach of SQL standard amounts to a very simple sufficient condition for consistency: the `SELECT` clause can only use columns listed in the `GROUP BY` clause.

Let us now return to the implicit group-by, and list some basic properties we expect from a reasonable semantics.

Natural. If no return expression is mixed, we follow the natural semantics: environments are grouped by the values of all return expressions with unaggregated identifier references.

Generic. If two environments agree over all identifiers used in the return expressions, they should end up in the same group.

Uniform. Whether two given environments belong to the same group does not depend on V. That is, based on $e_1, e_2, \ldots, e_k$, the semantics actually determines a partition of the whole $\mathbb{D}^{\mathsf{Ide}}$; for a given finite $V \subseteq \mathbb{D}^{\mathsf{Ide}}$ it takes the *induced* partition, with the groups obtained by intersecting V with the groups of $\mathbb{D}^{\mathsf{Ide}}$.

Syntax-insensitive. The semantics depends on the interpretation of the return expressions, not their syntactic representation. That is, if $[\![e_i]\!]^{\eta}_{U} = [\![e'_i]\!]^{\eta}_{U}$ for all η, U, and $i = 1, 2, \ldots, k$, then for each set V of environments, the answer returned for $e_1, e_2, \ldots, e_k$ should be the same as for $e'_1, e'_2, \ldots, e'_k$.

Obviously, to be of any use, the semantics also should be computable—hopefully, efficiently. This depends heavily on the domain and the available functions, and we postpone a more detailed discussion till Section 8.

What choice do we have then? Grouping environments by the values of all identifiers used in the return expressions leads to a semantics that is generic, uniform, and syntax-insensitive, but not natural. For instance, for the query in Example 2.2 the result would consist of two copies of the tuple (3, 1), while the natural semantics gives a single copy of the tuple (3, 2). We need a way to determine coarser partitions, taking into account the fact that different values of identifiers can lead to the same value of expressions. But we must be careful. For instance, we cannot simply take any consistent partition such that for each group the computed tuple of values is different.

Example 3.1. Consider the query below.

```
unwind [{a:1}, {a:2}, {a:3}] as x
return count(*)
```

In the reference implementation it returns a single tuple, as expected. But if the only requirement is the lack of repetitions in the answer, any partition into two groups is also acceptable, because these groups will necessarily have different sizes and will yield different rows in the answer.

```
+----------+     +----------+
| count(*) |     | count(*) |
+----------+     +----------+
| 3        |     | 2        |
+----------+     | 1        |
                 +----------+
```

Consequently, a semantics that takes any consistent partition that guarantees unique values for each group would be ill-defined, and only some choices would agree with the natural semantics.

4 Canonical Semantics?

Let us try a more systematic approach. Given V, which of its consistent partitions are more intuitive? It seems justified to expect that environments are put into different groups only if putting them together leads to inconsistencies. In other words, the coarser the partition, the better. That is, we use the ordering in which $\langle V_1, V_2, \ldots, V_n\rangle \leq \langle W_1, W_2, \ldots, W_m\rangle$, if each W_i is contained in some V_j. Because we are comparing two partitions of the same set, this is equivalent to saying that $V_1, V_2, \ldots, V_n$ can be obtained from $W_1, W_2, \ldots, W_m$ by merging some parts. In this order, the partition with a single group is the least (the coarsest), and the partition into singletons is the greatest (the finest). The coarser the partition, the harder for it to be consistent. The partition into singletons is consistent with each return expression, the partition with a single group will most often not be consistent.

In the absence of mixed expressions, there always exists the coarsest consistent partition: this is the partition chosen by the natural semantics! In fact, this a likely reason why this case is not controversial.

Lemma 4.1. *Consider expressions $e_1, e_2, \ldots, e_k$ without aggregating functions and $e'_1, e'_2, \ldots, e'_\ell$ without unaggregated identifier references. For every finite set V of environments, the partition of V according to the values of $e_1, e_2, \ldots, e_k$ is the coarsest partition of V that is consistent with $e_1, e_2, \ldots, e_k$ and $e'_1, e'_2 \ldots, e'_\ell$.*

Proof. Let $V_1, V_2, \ldots, V_n$ be the partition of V according to the values of $e_1, e_2, \ldots, e_k$. By definition, this partition is consistent with the expressions $e_1, e_2, \ldots, e_k$. Because the expressions $e'_1, e'_2, \ldots, e'_\ell$ do not contain unaggregated identifier references, each partition is consistent with them. Thus, $V_1, V_2, \ldots, V_n$ is consistent with all return expressions.

Moreover, it is at least as coarse as every consistent partition. Indeed, if a partition $W_1, W_2, \ldots, W_m$ is consistent with $e_1, e_2, \ldots, e_k$, then we can compute the values of these expressions for each W_i. These values indicate a group V_j that contains W_i. □

When mixed expressions are allowed, there may be multiple coarsest consistent partitions.

Example 4.2. Consider the following query.

```
unwind [{a:0,b:0}, {a:1,b:0}, {a:1,b:1}] as x
return x.a * max(x.b), count(*)
```

For this query, the partition into a single group

```
[{a:0, b:0}, {a:1, b:0}, {a:1, b:1}]
```

is not consistent, because the expression `max(x.b)` evaluates to 1 and the value of the return expression is different for the first two environments. On the other hand, the partition into

`[{a:0, b:0}, {a:1, b:0}]` and `[{a:1, b:1}]`

is consistent, and so is the partition into

`[{a:0, b:0}]` and `[{a:1, b:0}, {a:1, b:1}]`.

Hence, we have two coarsest consistent partitions. To make things even worse, they lead to different answers.

```
+----------------------------+
| x.a * max(x.b) | count(*) |
+----------------------------+
| 0              | 2         |
| 1              | 1         |
+----------------------------+
+----------------------------+
| x.a * max(x.b) | count(*) |
+----------------------------+
| 0              | 1         |
| 1              | 2         |
+----------------------------+
```

This example shows that in the presence of mixed expressions it will be difficult to design a semantics that feels canonical. Let us shift the perspective a bit and see if the uniformity assumption helps. In uniform semantics, we have an underlying partition of the set $\mathbb{D}^{\mathsf{Ide}}$ of all environments. Such a partition is consistent, if each finite subset of each of its parts is consistent with all return expressions. The natural semantics without mixed expressions is uniform, and the underlying partition of $\mathbb{D}^{\mathsf{Ide}}$ is the coarsest consistent partition (the argument is analogous to the one in Lemma 4.1). For the query in Example 4.2 there are still at least two coarsest consistent partitions of $\mathbb{D}^{\mathsf{Ide}}$. This is easiest to see for $\mathbb{D} = \{0, 1\}$, because the only additional environment to consider is `{a:0, b:1}`, but it is more instructive to consider a larger domain.

Example 4.3. Let $\mathbb{D} = \mathbb{N}$. Let us examine the coarsest consistent partitions of $\mathbb{D} = \mathbb{N}$ for the return values of the query in Example 4.2. If a consistent group contains an environment binding b to a non-zero value, then all environments in this group must agree on a. Hence, in any coarsest consistent partition of $\mathbb{D}^{\mathsf{Ide}}$, environments binding non-zero values to b will be partitioned according to the value bound to a. Let Z_i be the part containing the environments binding a to i. For each environment η binding b to 0 we have a choice: we can add η either to $Z_{\eta(a)}$, or to an additional part Z containing only environments binding b to 0. Thus, each choice of a set Z of environments binding b to 0 leads to a different coarsest consistent partition of $\mathbb{D}^{\mathsf{Ide}}$. Interestingly, none of these partitions guarantees that for each finite $V \subseteq \mathbb{D}^{\mathsf{Ide}}$ the partition induced on V is a coarsest consistent partition. Indeed, depending on whether the environment binding a and b to 0 is in Z or in Z_0, either `[{a:0, b:0}, {a:0, b:1}]` or `[{a:0, b:0}, {a:1, b:0}]` will be unnecessarily split into two parts.

Thus, even among uniform semantics it will be difficult to single out a canonical one.

5 Greedy Semantics

A consistent partition is nothing more than a collection of ad-hoc choices that happens to lead to a well-defined answer. The notion of consistency is very sensitive to these choices. In the non-uniform variant, a group can be consistent because it contains some environment. For instance, consistency of a group may rely on the fact that the maximal value bound to some identifier is at least 1. This means that a subset of a consistent group need not be consistent, and even the intersection of two consistent groups need not be consistent. The uniform variant is more robust, because for each part we require that all its finite subsets are consistent. Still, we can rely heavily on the absence of certain environments in a group. For instance, in Example 4.3, we rely on b being always bound to 0 in Z, even though b is only used under aggregation.

Let us try to devise a notion of consistency that is less sensitive to the presence or absence of other environments in a group. Let us fix the return expressions $e_1, e_2, \ldots, e_k$. Under what conditions can two environments η and η' be put into the same group, regardless of other members in the group? Clearly, as soon as for each $e \in \{e_1, e_2, \ldots, e_k\}$,

$$[\![e]\!]^{\eta}_{U} = [\![e]\!]^{\eta'}_{U} \quad \text{for each finite } U \text{ containing } \eta \text{ and } \eta'.$$

In such case we say that η and η' are *pairwise consistent*, and write $\eta \sim \eta'$. Pairwise consistency need not be an equivalence relation but with some care we can use it to define a consistent partition of any given finite set of environments.

For a given $V \subseteq \mathbb{D}^{\mathsf{Ide}}$, let $\sim_V$ be the transitive closure of the relation obtained by restricting $\sim$ to V. Because $\sim$ is reflexive and symmetric, so is its restriction to V, and consequently the relation $\sim_V$ is an equivalence relation for each V.

Lemma 5.1. *The set of equivalence classes of $\sim_V$ is a consistent partition of V.*

Proof. Let us take η and η' from the same abstraction class Z of $\sim_V$. We will show that $[\![e]\!]^{\eta}_{Z} = [\![e]\!]^{\eta'}_{Z}$ for each return expression $e \in \{e_1, e_2, \ldots, e_k\}$. Because $\eta \sim_V \eta'$, there exist $\ell \in \mathbb{N}$ and $\eta = \eta_0,\ \eta_1,\ \ldots,\ \eta_\ell = \eta'$ in V such that for all $i < \ell$, for each return expression e, $[\![e]\!]^{\eta_i}_{U} = [\![e]\!]^{\eta_{i+1}}_{U}$ for all finite U containing η_i and η_{i+1}. Note that we also have $\eta_0 \sim_V \eta_1 \sim_V \cdots \sim_V \eta_\ell$, so all these environments are in Z. Hence, $[\![e]\!]^{\eta_0}_{Z} = [\![e]\!]^{\eta_1}_{Z} = \cdots = [\![e]\!]^{\eta_\ell}_{Z}$ and we are done. □

Thus, we can base a semantics on grouping according to $\sim_V$. We call it the *greedy semantics*. Let us see how it works on concrete examples.

Example 5.2. Let us consider the return expressions of the query from Example 4.2, with $\mathbb{D} = \mathbb{N}$. Two environments are pairwise consistent iff they agree over a. Hence, in this case $\sim$ is an equivalence relation, and the greedy semantics simply partitions each V into the equivalence classes of $\sim$. For the query in Example 4.2 the chosen partition would be `[{a:0, b:0}]`, `[{a:1, b:0}, {a:1, b:1}]`. Interestingly, the greedy semantics breaks the symmetry and chooses one of the two coarsest consistent partitions.

Example 5.3. Let us replace `max` by `min` in the query from Example 4.2.

```
unwind [{a:0,b:0}, {a:1,b:0}, {a:1,b:1}] as x
return x.a * min(x.b), count(*)
```

Assuming that $\mathbb{D} = \mathbb{N}$, two environments are pairwise consistent with the return expressions of this query iff they agree over a or at least one of them binds b to 0. This is not an equivalence relation, so we apply the transitive closure. As long as V contains an environment η_0 binding b to 0, as in the query above, the resulting partition consists of a single group, because $\eta' \sim \eta_0$ and hence $\eta' \sim_V \eta''$ for all $\eta', \eta'' \in V$. For the query above the answer is a single tuple $(0, 3)$. If all environments in V bind b to non-zero values, V will be grouped by the value of a. Notice that in both cases we get the coarsest consistent partition (as defined in the previous section), but these partitions are not induced by a single partition of $\mathbb{D}^{\mathsf{Ide}}$, because they do not agree with each other. Hence, the greedy semantics is not uniform.

Lemma 5.4. *The greedy semantics is natural, generic, and syntax-insensitive.*

Proof. Because the greedy semantics is defined in terms of the interpretation of the return expressions, it follows immediately that it is syntax-insensitive. If η and η' agree over all identifiers used in return expressions, then for each return expression e we have $[\![e]\!]^{\eta}_{U} = [\![e]\!]^{\eta'}_{U}$ for all finite U. Hence, $\eta \sim \eta'$, and consequently $\eta \sim_V \eta'$ for all finite V. This shows that the greedy semantics is generic. Finally, let us fix a sequence of unmixed return expressions. For expressions e without unaggregated identifier references, $[\![e]\!]^{\eta}_{U} = [\![e]\!]^{\eta'}_{U}$ for all η, η' and U; that is, such expressions are irrelevant for $\sim$. For expressions e without aggregating functions, the value does not depend on U. Hence, $\eta \sim \eta'$ iff $[\![e]\!]^{\eta} = [\![e]\!]^{\eta'}$ for all return expressions e with identifier references. The latter is exactly the grouping condition in the natural semantics. Moreover, $\sim$ is an equivalence relation, so $\sim_V$ coincides with $\sim$ for all finite V. This means that, in the absence of mixed expressions, the greedy and the natural semantics choose the same partition. That is, the greedy semantics is natural. □

6 Functional Semantics

The greedy semantics enjoys all postulated properties except uniformity, as shown by Example 5.3. The reason is that in order to make the resulting partition consistent, we compute the transitive closure of $\sim$ in V, not in $\mathbb{D}^{\mathsf{Ide}}$. A way to make the sematics uniform is to modify the notion of pairwise consistency as follows. Let us fix the return expressions $e_1, e_2, \ldots, e_k$. We say that η and η' are *strongly pairwise consistent* if for each return expression $e \in \{e_1, e_2, \ldots, e_k\}$,

$$[\![e]\!]^{\eta}_{U} = [\![e]\!]^{\eta'}_{U} \quad \text{for each finite } U.$$

This is an equivalence relation and we can use it directly to partition any subset of $\mathbb{D}^{\mathsf{Ide}}$ into equivalence classes. Having seen the greedy semantics first, one may find this modification unjustified. But in fact the resulting semantics amounts to grouping environments by the values of all return expressions, except that the expressions are only partially evaluated: the resulting objects are not elements of $\mathbb{D}$, but functions mapping finite sets of environments to $\mathbb{D}$. This is why we call it the *functional semantics.*

Example 6.1. In the query from Example 4.2, the environments η will get grouped according to the meaning of functions

$$f_\eta(U) = \eta(a) \cdot \max_{\eta' \in U} \eta'(b) \quad \text{and} \quad g_\eta(U) = |U| \,.$$

For the environment `{a:0, b:0}`, the first function maps each U to 0; for `{a:1, b:0}` and `{a:1, b:1}`, it maps each U to $\max_{\eta' \in U} \eta'(b)$. The second function is the same for all η and does not affect the partition. We end up with the partition into `[{a:0, b:0}]` and `[{a:1, b:0}, {a:1, b:1}]`, just like under the greedy semantics (see Example 5.2).

Lemma 6.2. *The functional semantics is natural, generic, uniform, and syntax-insensitive.*

Proof. The functional semantics is uniform by construction. The remaining properties are checked like in Lemma 5.4. □

The functional semantics is strictly finer than the greedy semantics. Indeed, two environments are in the same group under the functional semantics iff they are strongly pairwise consistent. But then they are also pairwise consistent and are in the same group under the greedy semantics, too. Strictness follows from the example below.

Example 6.3. As we have seen, under the greedy semantics, the query in Example 5.3 returns a single row. Under the functional semantics, we put η and η' into the same group only if

$$\eta(a) \cdot \min(B) = \eta'(a) \cdot \min(B)$$

for all finite $B \subseteq \mathbb{N}$. This happens iff $\eta(a) = \eta'(a)$. That is, under the functional semantics the considered query returns two rows, $(0,1)$ and $(0,2)$, corresponding to the partition into `[{a:0, b:0}]` and `[{a:1, b:0}, {a:1, b:1}]`.

7 Weak Functional Semantics

We believe that the functional semantics is the right semantics for implicit group-by from the theoretical point of view, but it may be impractical: aggregating functions are quite complex objects and the way they interact with each other makes it highly nontrivial to understand what a specific query does. A way to make it easier is to abstract away from the interpretation of aggregating functions. This can be done on several levels. For instance, we may want to remain aware of the fact that `avg(a)` is always equal to `sum(a)` divided by `count(a)`, or that `count(*)` always returns an integer. Here, we choose the fully opaque model: no knowledge about aggregating functions will be available to the semantics.

The idea is to replace each call of an aggregating function in expression e by a fresh variable. Because we want the semantics to be syntax-insensitive, equivalent calls should be replaced with the same variable. But when are two calls equivalent? Clearly, if the same function symbol and syntactically the same argument is used. But also when the arguments are syntactically different but semantically equivalent. That is, we consider $F(e)$ and $F'(e')$ equivalent iff

$$F = F' \quad \text{and} \quad [\![e]\!]^\eta = [\![e']\!]^\eta \quad \text{for each } \eta$$

(recall that aggregating function symbols are not nested, so e and e' do not contain them). Depending on the domain and the available functions, equivalence may be very hard, even undecidable; we discuss it in detail in Section 8.

Let X be a countably infinite set of variables. For each return expression e we define the expression $\tilde{e}$ over $\mathsf{Fun} \cup \mathsf{Ide} \cup X$ in which all occurrences of equivalent calls of aggregating functions are replaced with a single fresh variable from X. The way $\tilde{e}$ is obtained is complicated in general, but in practice aggregating function calls are most often of the form $F(a)$ for some identifier a. If this is the case, we simply replace each occurrence of $F(a)$ with a fresh variable $x_{F(a)}$. The value $[\![\tilde{e}]\!]^{\eta,\alpha}$ of a term $\tilde{e}$ with variables for a given environment η and a valuation α of variables is defined by induction in the usual way:

$$\begin{aligned} [\![a]\!]^{\eta,\alpha} &= \eta(a)\,, \\ [\![x]\!]^{\eta,\alpha} &= \alpha(x)\,, \\ [\![f(\tilde{e}_1, \tilde{e}_2, \ldots, \tilde{e}_p)]\!]^{\eta,\alpha} &= \hat{f}\big([\![\tilde{e}_1]\!]^{\eta,\alpha}, [\![\tilde{e}_2]\!]^{\eta,\alpha}, \ldots, [\![\tilde{e}_p]\!]^{\eta,\alpha}\big)\,, \end{aligned}$$

for $a \in \mathsf{Ide}$, $x \in X$, $f \in \mathsf{Fun}$, and terms $\tilde{e}_i$ with variables.

Let us fix the return expressions $e_1, e_2, \ldots, e_k$. In the *weak functional semantics* we put environments η and η' into the same group if for each return expression $e \in \{e_1, e_2, \ldots, e_k\}$,

$$[\![\tilde{e}]\!]^{\eta,\alpha} = [\![\tilde{e}]\!]^{\eta',\alpha} \quad \text{for all } \alpha : X \to \mathbb{D}\,.$$

This amounts to grouping by the values of the expressions $\tilde{e}_1, \tilde{e}_2, \ldots, \tilde{e}_k$, except that they are only partially evaluated, with the resulting objects being functions mapping valuations of variables into elements of $\mathbb{D}$.

Notice that the resulting partition is consistent with the return expressions: if the functions are identical for all environments in the group, they will give the same value when fed with the valuation that maps each variable to the value returned by any corresponding aggregating function call. It does not matter which of the equivalent calls associated to a given variable we choose, because equivalent calls always return identical values.

Example 7.1. Consider again the query from Example 4.2. To determine the answers under the weak functional semantics, let us replace `max(x.b)` with a variable ξ, and `count(*)` with a variable ζ. Then, the environments η should be grouped according to the meaning of functions

$$f_\eta(\xi) = \eta(a) \cdot \xi \quad \text{and} \quad g_\eta(\zeta) = \zeta\,.$$

For the environment `{a:0, b:0}`, the first function maps each ξ to 0; for `{a:1, b:0}` and `{a:1, b:1}`, it maps each

ξ to itself. The second function is the same for all η and does not affect the partition. We end up with the partition into `[{a:0, b:0}]` and `[{a:1, b:0}, {a:1, b:1}]`, just like under the greedy semantics and the functional semantics (see Examples 5.2 and 6.1).

The weak functional semantics may not seem much simpler than the functional semantics: in both cases we are comparing functions. The difference is that before the functions were taking a set of environments as input, and now they take a tuple of elements of $\mathbb{D}$. That is, the problem amounts to testing equivalence of expressions with variables. As we have mentioned, this may be very hard in general, but in practice these expressions would be very simple and the semantics should be predictable.

The weak functional semantics behaves quite reasonably. Because we are abstracting away from the interpretation of aggregating functions, there is no hope for the semantics to be syntax-insensitive, but we have the following weaker property instead. A semantics for implicit group-by is *weakly syntax-insensitive* if whenever $[\![\widetilde{e_i}]\!]^{\eta,\alpha} = [\![\widetilde{e_i'}]\!]^{\eta,\alpha}$ for all η, α, and $i = 1, 2, \ldots, k$, the answer returned for $e_1, e_2, \ldots, e_k$ is the same as for $e_1', e_2', \ldots, e_k'$, for each set V of environments. Note that this weaker property also excludes the unexpected behaviour of the reference implementation revealed by Examples 2.4 and 2.5, guaranteeing that the two queries give the same answers.

Lemma 7.2. *The weak functional semantics is natural, generic, uniform, and weakly syntax-insensitive.*

Proof. That the semantics is uniform and weakly syntax-insensitive follows directly from the definition. Genericity is immediate as well: if η and η' agree over all identifiers used in the return expressions, then $[\![e]\!]^{\eta,\alpha} = [\![e]\!]^{\eta',\alpha}$ for all α for each return expression e. Let us check that the semantics is natural. Assume that all return expressions are not mixed. For expressions e without unaggregated identifier references, $[\![\tilde{e}]\!]^{\eta,\alpha}$ does not depend on the environment η; such expressions will not influence the grouping. For expressions e without aggregating functions we have $\tilde{e} = e$. Consequently, $[\![\tilde{e}]\!]^{\eta,\alpha} = [\![\tilde{e}]\!]^{\eta',\alpha}$ iff $[\![e]\!]^{\eta} = [\![e]\!]^{\eta'}$. This gives precisely the grouping condition of the natural semantics without mixed expressions. □

Let us note that the weak functional semantics is strictly finer than the functional semantics.

Lemma 7.3. *The weak functional semantics is at least as fine as the functional semantics.*

Proof. Assume η and η' are put into different groups by the functional semantics. Then, $[\![e]\!]^{\eta}_U \neq [\![e]\!]^{\eta'}_U$ for some return expression e and some finite U. Let α_U be obtained by valuating each variable in $\tilde{e}$ according to the value of the associated aggregating function calls over U. Because all calls associated to a single variable return the same value, α_U is well defined. By definiton of $\tilde{e}$, $[\![\tilde{e}]\!]^{\eta,\alpha} = [\![e]\!]^{\eta}_U \neq [\![e]\!]^{\eta'}_U = [\![\tilde{e}]\!]^{\eta',\alpha}$. Thus, η and η' are put into different groups by the weak functional semantics as well. □

Strictness is shown by the following example.

Example 7.4. Consider the following query.

```
unwind [{a:1}, {a:2}] as x
return x.a * sum(0)
```

The functional semantics is aware of the fact that `sum(0)` always evaluates to 0 and so does the whole return expression. Therefore, it returns a single row with value 0. The weak functional semantics knows nothing about the value of `sum(0)`, except that it is always equal to the value of, say, `sum(x.a - x.a)`. Hence, it returns two rows with value 0.

For the example above the weak functional semantics gives an answer less intuitive than the one given by the functional semantics. We believe this will actually be the case always when the two semantics differ and the queries are simple enough to be understood. On the other hand, under the weak functional semantics the answers should be in general easier to predict.

8 Computability and Feasibility

One key aspect we have not discussed yet is computability. Semantics of query languages are generally expected to be simple enough to be obviously computable. But in the case of the implicit group-by construction, this is not the case. As we have seen, all three proposals are rather involved mathematically and it seems unavoidable without compromising the postulates. Computability becomes a serious issue.

First, we need to adjust the abstract setting of the previous sections to the computational reality. For a start, we shall work in the real domain. Formally, $\mathbb{D} = \mathbb{R} \cup \{\bot\}$, where $\bot$ indicates that the function is undefined, like $1/x$ for $x = 0$, and it propagates through all functions. Otherwise, functions are interpreted as the actual mathematical functions over the reals, abstracting away from the fact that in reality approximate implementations are used. This determines the three semantics for all finite $V \subseteq (\mathbb{R} \cup \{\bot\})^{\mathsf{Ide}}$, but we are only interested in computing it for $V \subseteq \mathbb{Q}^{\mathsf{Ide}}$.

The computational essence of the semantics is the following *term equivalence problem*: given two terms built out of variables, rationals, and functions from `Fun`, decide if they define the same multivariate function over the reals. If the set `Fun` of available functions contains only `+`, `-`, `*`, `/`, `abs()`, decidability of this problem follows from Tarski's decidability of the first-order theory of the reals (its existential fragment, in fact) [2, 22]. Allowing `exp()` is problematic: whether the first-order theory of the reals extended with the exponential function is decidable, is a question asked already by Tarski, and still remaining open. As shown by Macintyre and Wilkie [13], this question is closely related to Schanuel's conjecture, a famous open problem in transcendental number theory. While it is known that Tarski's question amounts to determining if two given expressions are equal for some values of variables, it is does not necessarily reduce to the term equivalence problem, where we ask if the expressions are equal for all values. Nevertheless, the term equivalence problem becomes undecidable rather quickly: already for expressions using integers, `+`, `*`, `abs()`, `sin()`, and a single variable. This result was initially obtained for a larger class of functions including additionally `exp()` and the constants π and $\ln(2)$ [19]. After Hilbert's 10th problem was shown to be undecidable [14], the need for `exp()` and $\ln(2)$ was eliminated [3]. Recently, π has been eliminated as well [12]: as formulated, the result is about determining if there exists $z \in \mathbb{R}$ such that $e(z) > 0$, but in the presence of `abs()` this is the same as non-equivalence of terms $\mathtt{abs}(e(z))$ and $-e(z)$.

The undecidability of term equivalence propagates to all three proposed semantics.

Corollary 8.1. *If return expressions can use integers,* `+`, `*`, `abs()`, `sin()`, *and either* `avg()` *or* `sum()`, *then the greedy semantics, the functional semantics, and the weak functional semantics are not computable.*

Proof. It is straightforward to reduce the term equivalence problem to the evaluation of implicit group-by queries. Let $e_1(z)$ and $e_2(z)$ be the input expressions. Consider the following query using the expression $e(z) = e_1(z) + -1 \cdot e_2(z)$.

```
unwind [{a:0}, {a:1}] as x
return x.a * e(avg(x.a))
```

We claim that under each of the three semantics, this query returns one row iff $e(z) = 0$ for all z. For the weak functional semantics, this is by definition. Under the functional semantics, the query returns one row iff for each finite multiset A of reals, $e(\mathtt{avg}(A)) = 0$. Because A can be any singleton, the latter is equivalent to the assertion that $e(z) = 0$ for all $z \in \mathbb{R}$. Finally, under the greedy semantics, the query returns one row iff for each finite multiset A of reals that contains 0 and 1, $e(\mathtt{avg}(A)) = 0$. This time it is enough to observe that for each $z \in \mathbb{R}$ it holds that $\mathtt{avg}(\langle 0, 1, 3z - 1\rangle) = z$. Exactly the same arguments work with `avg()` replaced by `sum()`, except that in the last case we use $z - 1$ rather than $3z - 1$. □

The decidability result propagates as well, but only for the weak functional semantics.

Corollary 8.2. *If return expressions can use only rationals,* `+`, `-`, `*`, `/`, `abs()`, *and arbitrary aggregating functions, then the weak functional semantics is computable.*

Proof. In order to perform an implicit group-by operation with respect to return expressions $e_1, \dots, e_k$ under the weak functional semantics, we first compute $\tilde{e}_1, \dots, \tilde{e}_k$. To do this, for each two calls $F(e)$ and $F(e')$ of the same aggregating function F in the same return expression, we test if $[\![e]\!]^\theta = [\![e']\!]^\theta$ for each $\theta \in \mathbb{R}^{\mathsf{Ide}}$. This is an instance of the term equivalence problem for expressions using rationals and `+`, `-`, `*`, `/`, `abs()`; we solve it by reduction to the first-order theory of the reals. Then, we replace equivalent calls with the same fresh variable. Having computed $\tilde{e}_1, \dots, \tilde{e}_k$, we check for each pair of environments η, η' from V if for all i, $[\![\tilde{e}_i]\!]^{\eta,\alpha} = [\![\tilde{e}_i]\!]^{\eta',\alpha}$ for all α. Substituting identifiers in $\tilde{e}_i$ with the values according to η and η', we get another instance of the term equivalence problem and solve it like before. □

The existential theory of the reals is intractable: in PSPACE and NP-hard. Practical algorithms can handle expressions of size expected in typical queries, but possibly not well enough for an industry-grade DBMS. Moreover, the procedure sketched above is quadratic in the size of the data and its nature seems to exclude the use of standard grouping techniques based on sorting or hashing. Consequently, the weak functional semantics under these syntactic restrictions is unlikely to be applied in practice. But if we additionally forbid `abs()` and `/`, we get in the range of implementable solutions.

Lemma 8.3. *Given an environment η and return expressions using rationals,* `+`, `-`, `*`, *and arbitrary aggregating functions, one can compute in time $k \cdot \mathrm{poly}(\ell, 2^r)$ a key κ_η such that grouping any set of environments by these keys realizes the weak functional semantics; here, ℓ is the maximal number of bits used to represent any number in the input, r is the maximal size of return expressions, k is the number of return expressions.*

Proof. We first compute $\tilde{e}$ for each return expression e. This time the arguments of the aggregating function calls are polynomials in Ide with rational coefficients. Their equivalence can be tested by comparing their canonical representations as simplified sums of monomials ordered lexicographically. These representations are unique: two polynomial functions are identical over $\mathbb{R}$ if they have identical representations. We can compute these representations in time $\mathrm{poly}(L, 2^r)$ by standard algebraic simplification. The resulting expression $\tilde{e}$ is a polynomial $p(\bar{x})$ in variables $\bar{x}$ and identifiers from Ide with rational coefficients. Let us write $p^\eta(\bar{x})$ for the polynomial obtained from $p(\bar{x})$ by substituting the elements of Ide by the values bound to them by η (always rational). The key κ_η is obtained by concatenating the canonical representations of $p^\eta(\bar{x})$ obtained for all return expressions e. □

Using the keys given by Lemma 8.3, we can perform the grouping using hashing or sorting with respect to any linear order, for instance, lexicographic order over binary encodings. The sorting algorithm gives the following worst-case upper bound.

Corollary 8.4. *If return expressions use only rationals,* `+`, `-`, `*`, *and arbitrary aggregating functions, then grouping under the weak functional semantics can be carried out in time $k \cdot \mathrm{poly}(\ell, 2^r) \cdot N \cdot \log N$, where ℓ is the maximal number of bits used to represent any number in the input, r is the maximal* *size of return expressions, k is the number of return expressions, and N is the number of input environments.*

The exponential dependence on the size r of return expressions is purely theoretical. Return expressions in queries are typically so simple that the resulting polynomials $p(\bar{x})$ will be very small: as variables correspond to different aggregating function calls, a reasonable assumption is that these polynomials will have just a couple of variables, very low degree, and not too large coefficients; coefficients in $p^\eta(\bar{x})$ may be larger, as they depend on the values used in η. Note also that keys are computed separately for each environment, which makes this part of the procedure highly parallelizable.

There is also a natural hashing approach, in the spirit of randomized polynomial identity testing. Instead of canonical representations of polynomials $p^\eta(\bar{x})$, we can take the values of these polynomials in several randomly chosen points (the same for each η). Environments with different values of this hashing function will necessarily end up in different groups, so it suffices to perform grouping among environments with the same value of the hashing function. This is done like before, using the canonical representation of polynomials.

A natural question is what happens if we work over rationals, not over reals. This seems to be a reasonable choice: once we have eliminated functions like `sin()` and `exp()`, and only kept basic arithmetic operations, we will never get outside of $\mathbb{Q}$ if environments use only values from $\mathbb{Q}$. Because integers are first-order interpretable in the field of rationals [21], the first-order theory of rationals is undecidable. Can we decide term equivalence? When expressions use rationals, `+`, `-`, `*`, and `/`, term equivalence is at least as hard as a long standing open problem: Given a multivariate polynomial with integer coefficients, does it have a rational zero. Indeed, $\frac{p(\bar{x})}{p(\bar{x})}$ is equivalent to 1 over $\mathbb{Q}$ iff $p(\bar{x})$ has no rational zeros. Arguing like in Corollary 8.1, we see that computing any of the three semantics is at least as hard as solving this problem.

If we forbid division and allow only rationals, `+`, `-`, `*`, and arbitrary aggregating functions, we end up with polynomials again. They can be still used as keys, because two multivariate polynomials are equal over rational numbers iff they are identical. (The same holds for natural numbers, or actually any infinite set of numbers.) In particular, the semantics over rationals coincides with the semantics over reals. Once we have agreed to work over rationals, we can ease the syntactic restrictions a little. Let us allow all operations and functions available in the syntax, as long as they are not applied to aggregating function calls or inside them. Then, for each η we obtain a polynomial $p^\eta(\bar{x})$ over $\bar{x}$ whose coefficients are expressions build out of rational values and mathematical functions. In the final answer these expressions are to be evaluated, not processed symbolically. Hence, we propose to evaluate them immediately using the available approximate implementations of mathematical functions and to proceed as before with the resulting polynomials with rational coefficients. Formally, this corresponds to changing the interpretation of functions from the actual mathematical meaning to the implemented approximate meaning.

What about integers and natural numbers? Limiting the whole domain this way may be too restrictive, but suppose that we would like to enrich the weak functional semantics with some knowledge about aggregating functions. For example, one could try to make the semantics aware of the fact that `count()` always returns a natural number. This would be reflected in the definition of the semantics by allowing some of the variables in the expression $\tilde{e}$ to take only natural numbers as values. Can we test equivalence of such expressions? The first-order theory of natural numbers is of course undecidable. Term equivalence is also undecidable as long as expressions can use integers, `+`, `-`, `*`, `/`, and arbitrary aggregating functions. Indeed, for any polynomial $p(\bar{x})$ with integer coefficients, $\frac{p(\bar{x})}{p(\bar{x})}$ is equivalent to 1 over natural numbers iff $p(\bar{x})$ has no zeroes in natural numbers. The latter is one of the equivalent formulations of Hilbert's 10th problem, well known to be undecidable. Like in Corollary 8.1, it follows that the weak functional semantics is not computable. Without `/` we are in the setting of Lemma 8.3, and everything works like for $\mathbb{Q}$. For integers, the whole picture is identical.

9 Conclusion

Taking an inconsistency in the reference implementation as a starting point, we investigated ways of giving Cypher's implicit group-by a reasonable semantics. We have shown that for unmixed expressions there exists the coarsest semantics, but not so in the presence of mixed expressions. In our opinion, this explains why unmixed expressions are noncontroversial, whereas mixed expressions lack a canonical semantics.

While forbidding mixed expressions is the simplest solution, it is also a rather radical one. We have proposed a more flexible solution, going beyond unmixed expressions: the weak functional semantics for return expressions that apply only `+`, `-`, `*` to aggregating function calls and inside them. This is a clean semantics for a natural syntactic fragment, very well suited for implementation.

The proposed solution would become even more attractive if one could extend the set of allowed operations with `/`, without compromising feasibility. This would require extending Lemma 8.3, which seems challenging, but there are no immediate complexity-theoretic obstacles over the reals. As we have observed, over the rationals even decidability is highly problematic: it is equivalent to the decidability of the existence of rational zeroes for a given polynomial with integer coefficients, which is a long standing open problem.

To what extent the proposed solution could be upgraded to the functional semantics or the greedy semantics is another open question, well worth further investigation.

References

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proc. SIGMOD 2018*. 1421–1432. https://doi.org/10.1145/3183713.3190654

[2] John Canny. 1988. Some Algebraic and Geometric Computations in PSPACE. In *Proc. STOC 1988*. ACM, New York, NY, USA, 460–467. https://doi.org/10.1145/62212.62257

[3] B. F. Caviness. 1970. On Canonical Forms and Simplification. *J. ACM* 17 (1970), 385–396.

[4] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. 2018. Formal Semantics of the Language Cypher. *CoRR* abs/1802.09984 (2018). arXiv:1802.09984 http://arxiv.org/abs/1802.09984

[5] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proc. SIGMOD 2018*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[6] Markus P Fromherz. 1993. *A Methodology for Executable Specifications: Combining Logic Programming, Object-orientation and Visualization*. Ph.D. Dissertation. Department of Computer Science, University of Zurich.

[7] Norbert E. Fuchs. 1992. Specifications Are (Preferably) Executable. *Softw. Eng. J.* 7, 5 (Sept. 1992), 323–334. https://doi.org/10.1049/sej.1992.0033

[8] Gopal Gupta and Enrico Pontelli. 2002. Specification, Implementation, and Verification of Domain Specific Languages: A Logic Programming-Based Approach. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*. 211–239. https://doi.org/10.1007/3-540-45628-7_10

[9] Neo4j Inc. 2018. The GQL Manifesto. https://gql.today/

[10] ISO/IEC 9075:2016 2016. *Information technology – Database languages – SQL (ISO/IEC 9075:2016)*. Standard. International Organization for Standardization, Geneva, Switzerland.

[11] R. Kowalski. 1985. The Relation Between Logic Programming and Logic Specification. In *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 11–27. http://dl.acm.org/citation.cfm?id=3721.3722

[12] M. Laczkovich. 2003. The Removal of π from Some Undecidable Problems Involving Elementary Functions. *Proc. Amer. Math. Soc.* 131, 7 (2003), 2235–2240. http://www.jstor.org/stable/1194043

[13] Angus Macintyre and Alex J Wilkie. 1996. On the decidability of the real exponential field. In *Kreiseliana: About and around Georg Kreisel*, Piergeorgio Odifreddi (Ed.). A.K. Peters, 441–467.

[14] Yu. Matiyasevich. 1970. The Diophantineness of enumerable sets (Russian). *Dokl. Akad. Nauk SSSR* 191 (10 1970), 279–282.

[15] openCypher. 2017. The Summer of Syntax: Aggregation and grouping. http://www.opencypher.org/blog/2017/07/27/ocig1-aggregations-blog Accessed: 2018-09-24.

[16] openCypher. 2018. *Cypher Query Language Reference, Version 9*. Technical Report. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf

[17] openCypher. 2018. openCypher. http://www.opencypher.org/

[18] openCypher. 2018. The Cypher Technology Compatibility Kit. https://github.com/opencypher/openCypher/tree/master/tck

[19] Daniel Richardson. 1968. Some Undecidable Problems Involving Elementary Functions of a Real Variable. *J. Symbolic Logic* 33, 4 (12 1968), 514–520. https://projecteuclid.org:443/euclid.jsl/1183736504

[20] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph Databases*. O'Reilly Media, Inc.

[21] Julia Robinson. 1949. Definability and decision problems in arithmetic. *J. Symb. Log.* 14 (1949), 98–114.

[22] Alfred Tarski. 1948. *A Decision Method for Elementary Algebra and Geometry*. RAND Corporation, Santa Monica, CA, USA. iii+60 pages.