Controlled Language for Geographical Information System Queries

Sela Mador-Haim, Yoad Winter, and Anthony Braun

Technion I.I.T
selam@cs.technion.ac.il, winter@cs.technion.ac.il,
tonyb@geofocus.co.il

Abstract

Natural language interface to spatial databases have not received a lot of attention in computational linguistics, in spite of the potential value of such systems for users of Geographical Information Systems (GISs). This paper presents a controlled language for GIS queries, solves some of the semantic problems for spatial inference in this language, and introduces a system that implements this controlled language as a novel interface for GIS.

1 Introduction

Geographical Information Systems (GISs) are information systems for processing of data that pertains to spatial or geographic coordinates [14]. Even though GISs are enjoying a rapidly growing users community, the current systems are often difficult to use or require a long learning process [13]. In the GIS literature [15,16,5,8], it has been well-acknowledged that natural language interfaces (NLIs) would significantly enhance the exploitation of the more complex features of GISs, yet despite the potential value of NLIs for GISs, the work on this subject has so far been rather limited [16]. To the best of our knowledge, existing NLIs for GISs are limited in scope and expressive power and lack the ability to express complex relationships over spatial entities. Some works ([9,17,12]) have demonstrated NLIs using a database that contain geographically related data. Those databases, however, lack any actual spatial information (e.g. geometric polygons representing buildings), and therefore do not deal with the problem of inferring spatial relations from such representations.

In general, the design of NLIs to databases is regarded as a difficult problem since human interaction is often vague, ambiguous or highly contextualized [15,1]. The approach we take in this paper is to avoid many of these problems by designing a system that uses a controlled language for GIS queries. Such

controlled languages [10,11], which is based on a fragment of English, can be designed in a way that minimizes the use of vague, ambiguous and context-dependent expressions, while maintaining the ability to express very complex queries in a language that is a subset of English. We greatly benefit from the fact that GISs are a closed, well-defined domain, which enables us to focus on the data independent core of the language. We show that the addition of the data dependent portion can be done semi-automatically and requires very low effort.

Our implementation of an NLI for GISs involves four major tasks: first, the definition the data independent lexicon, which was done using simple applicative categorial grammar (Ajdukiewicz-Bar-Hillel calculus). Second, we develop a suitable semantic representation for GIS queries, which we call λSQL, and a method to translate natural-language queries via λSQL into spatially-enabled SQL. The third task is the definition of the semantics for spatial relations (esp. prepositions) in the lexicon in accordance with the intuitive understanding of such relations, which involves tackling certain aspects of spatial prepositions that where never dealt with before. The fourth task is the development of methods to add the data dependent portion of the lexicon with minimal effort, including an automatic tool that generates lexical entries from the actual geographical database in use.

The paper is organized as follows: Section 2 introduces λSQL and describes the translation scheme from natural language into SQL queries. Section 3 reviews the architecture of the lexicon. Section 4 discusses semantic issues concerning spatial relations in natural languages. Section 5 presents our implementation, and section 6 concludes.

2 A compositional approach for building SQL queries

SQL is a recursive language in the sense that it allows using an SQL query as a part of an expression inside another SQL query. However, due to its complex syntax, the construction of an SQL query in a compositional way from a query in natural language is far from being a straightforward task. One way to tackle this problem is by using an intermediate representation [4,10]. While such an intermediate language avoids the complications of composing SQL queries directly, its downsides are the additional translation phase it requires and the fact that such intermediate languages are usually not as expressive as the target language.

We introduce an intermediate representation language, which we call λSQL. This language only adds the necessary "compositional glue" to SQL. As a result, only a simple translation process is necessary to convert λSQL queries into normal SQL syntax. λSQL is basically expressions in the simply typed λ Calculus with the addition of syntactic sugar for SQL-like syntax.

The typical syntax of a *select* SQL-command for querying a database is:

SELECT < *selectlist* > FROM < *tablelist* > WHERE < *whereclause* >;

The *selectlist* parameter is usually a list of fields to be displayed, but it also allows other expressions such as aggregate functions (e.g. field summation). The *tablelist* parameter is a list of tables to query and *whereclause* is a boolean expression that restricts the rows in the query.

The syntax of λSQL is very close to that of an SQL *whereclause*, with the addition of λ operators. The atoms of λSQL are real numbers, strings and typed identifiers. The base types in λSQL are: t - Boolean, r - real numbers, str - strings, g - spatial data and e - entries in the database. These base types correspond to the base types that are found in GIS databases, with the addition of one additional type, e, for database entries (tuples). Expressions are built from atoms via function applications: $exp1(exp2)$, infix operators: $exp1\ op\ exp2$ and the operators $\lambda v.exp$ and $\exists v.exp$. The infix operators in λSQL correspond to SQL operators, and include Boolean AND/OR, arithmetic operators $(+, -, *, /)$ and comparators $(>, <, =, <=, >=, !=)$. One additional important operator in λSQL is the dot operator, as in $var.fieldname$, where var is of type e and $fieldname$ is a function from entries in the database to entities of a basic type (i.e. it is of type et, er or es). A dot expression is equivalent to $fieldname(var)$, a function that returns the value of a field of a given entry.

In general, the only two syntactic elements in λSQL that do not correspond directly to SQL syntax are the λ and $\exists$ operators. Translation from λSQL expressions to SQL queries is done by recursive traversal over the expression. During traversal, whenever certain patterns are recognized, these patterns are replaced by a corresponding SQL *select* statement. Each λ operator corresponds to a *select* statement, which can be nested inside another *select*. In addition to λ operators, three different synthetic elements may affect the translation pattern:

P1 A function over a λ expressions, as in $f(\lambda v.exp)$, is treated as an aggregate function.

P2 In the simplest pattern, the type of the variable v in λv is e, and it corresponds to a query that returns a set of entries. When the variables that the λ operator binds is of any other base type, the pattern: $\lambda x.\exists y.(x = expr1\ AND\ expr2)$ is expected, which is translated into *SELECT expr1 FROM layer WHERE expr2*.

P3 Any additional $\exists$ operator which is not part of the pattern above is translated as a table join (where *tablelist* parameter contains more than one query). For example, the expression $\lambda x_e.\exists y_e.(x.layer = \text{“}layer1\text{”}\ AND\ y.layer = \text{“}layer2\text{”}\ AND\ exp)$ is translated into: *SELECT x.* FROM layer1 AS x, layer2 AS y WHERE exp*. Each additional $\exists$ adds an additional table to the list.

The translation process is guaranteed to be successful due to constraints over the λSQL expressions in the lexicon that enforce conformity to the above patterns. As an example for λSQL, consider the following fragment from our lexicon is presented below:

Word	Category [1]	Semantics
buildings	N	$\lambda x_e.(x.layer_{es} = "building")$
with	$N\backslash N/N$	$\lambda n1_{et}.\lambda n2_{et}.\lambda x_e.(n1(x)\ AND\ n2(x))$
more than	Rs/R	$\lambda n_r.\lambda x_r.(x > n)$
two	R	2
floors	$N\backslash Rs$	$\lambda p_{rt}.\lambda x_e.p(x.floors_{er})$
highest	N/N	$\lambda n_{et}.\lambda x_e.(n(x)\ AND\ (x.height_{er} =$ $max_{(rt)r}(\lambda r_r.\exists y_e.(n(y)\ AND\ r = y.height_{er}))))$

The natural language expression "buildings with more than two floors" will be parsed into the λSQL expression: $\lambda x_e.(x.layer_{es} = "building"\ AND\ x.floors_{er} > 5)$. Note that while functional applications during parsing eliminated most λ operators, the λ operator that is introduced by the lexical entry for *buildings* is not eliminated. This remaining λx_e is used to describe a query over a variable x. In order to generate an SQL query, however, one additional piece of information is required: the name of a table to query. This information is provided via the *layer* keyword (layers, or feature sets in GIS terminology are equivalent to tables in general databases). While usually the $fieldname$ following the dot operator is a name for an actual field in the database (such as $floors$ in the above example), *layer* is a virtual attribute in λSQL, used to associate a layer with a variable. Whenever an expression such as $x.layer_{es} = "building"$ is found, the parser associates x with the table "building", and hence the above expression is translated into the SQL query:

```
SELECT x.* FROM building AS x WHERE x.floors>5;
```

A bit more complex example is the query "highest buildings", which is translated into: $\lambda x_e.(x.layer_{es} = "building''\ AND\ x.height_{er} = max_{(rt)r}(\lambda r_r.\ \exists y_e.(y.layer_{es} ='' building''\ AND\ r = y.height_{er})))$. This expression demonstrates several features of λSQL. Note that max is a free identifier, which is expected to be a name of an SQL function. The function max receives a λ expression, and therefore interpreted as an aggregate function. Finally, the

[1] The atomic categories in the lexicon correspond to the base types of λSQL, as well as sets of base type entities. For example, category R corresponds to type r and Rs corresponds to the type (rt).

expression in the argument of *max* fits pattern P2 above, and the result is:

SELECT x.* FROM building AS x WHERE x.floors=(SELECT max(y.floors) FROM building);

3 Lexicon architecture

The data independent part of the lexicon is the core of our controlled language. This is the part of the lexicon that involves general logical and spatial operators that do not depend on the actual GIS. By carefully selecting the data-independent lexical items, we are able to express very complex queries while avoiding vagueness and ambiguity problems that often undermine the usability of NLIs. An important part of our work is the ability to express spatial relations between GIS objects. However, non-spatial lexical items are an important part of the lexicon as well. In the first part of this section we describe the non-spatial items in the lexicon. In the following part we review the spatially-related lexical items. Finally we present classes of data-dependent lexical items.

3.1 Non-spatial lexical items

Non-spatial lexical items can be partitioned into the following groups:

- units, such as *meters*, *kilometers*, *miles*, *acres*. The lexical definition for these items converts any unit into standard units (e.g. metric units).
- numerical relations, such as *less than*, *at least*, *between n and m*. Numerical relations represent a set of real numbers.
- Superlatives: *biggest*, *smallest*, *most*, *least*. The words *most* and *least* can be used to refer to the maximal or minimal value of any numerical field in the database. Other words such as *largest* and *longest* are used as abbreviation for "most area" and "most length".
- Boolean connectives: *and*, *or*, *not*
- Other lexical entries: *that*, *which*, *is*, *are*, *with*, *without*, *have*.

3.2 Spatial lexical items

As mentioned before, we wish to design a controlled language that would avoid the pitfalls of vagueness and context-dependent ambiguity. In order to satisfy this requirement, we need to avoid vague qualitative relations such as *near, far* and *almost*. Another type of relations that need to be avoided are *projective* relations such as *in front of, behind, left* and *right*. The meaning of these prepositions involves context-dependent[6] elements that are hard to handle within a controlled language.

The following spatial relations are included in the lexicon:

- Intersectional relations, following Egenhofer's 9-intersection model [3]: *in, outside of, borders, overlaps, crosses, contains* and *intersects*. Note that only the first two expressions are prepositions, while the others are verbs.
- Distance: the word *from* is used to specify exact distance, as in "200m from a lake".
- Constructors: *intersection of, border of* and *center of*. These words are used to refer to spatial entities that do not exist in the database, but can be derived from existing objects. For example, assuming "42nd Street" and "Broadway" are objects in the database, "the intersection of 42nd street and Broadway" can be constructed by intersecting the geometrical representations of the two streets.
- Relative orientation: *north of, south east of* and the 3-place relation *between* are all used to describe the orientation of one object relative to another object (or objects, as in the case of *between*).
- Superlatives: *closest* and *furthest* are spatially-related superlatives.

3.3 Data-dependent lexical items

Data dependent lexical items are lexical items that refer to specific data inside the database and therefore change from one data set to another. GIS data are divided into separate thematic feature classes or layers, whereby each layer consists of one type of geometrical entity such as a building, street or utility pole. For each layer there is usually an associated set of attributes that represent non-spatial data attached to real world geometric objects. These may be boolean data, numeric data or strings. Examples for such attributes are the number of floors in a building or a street name. String values such as street names should be part of the lexicon as well.

An example for such a template is:

$$\text{"}\#strval\text{"}\ \ N/N\{l = \#layer\}\ \lambda n.\lambda x.(n(x)\ AND\ (x.\#attr\ like\ \#strval))$$

The "#strval" template defines lexical items that refer to strings inside the database. The lexical analyzer searches the database for strings that match lexical tokens that are not present in the lexicon. For each such string it finds, the above template is instantiated with the relevant layer name, attribute name and string value. Similar templates are used for layer names and attributes of various types. In case the lexical entries need to be different than the actual names in the database, a definition file is used to add those lexical items and instantiate the relevant templates for those items. No knowledge in λSQL is required in order to edit the definition file.

4 Semantics of spatial prepositions

While some progress was made in semantic theories of prepositional phrases in recent years [18,7], certain aspects of spatial linguistic phenomena have not been treated as extensively in the semantic literature, but are nevertheless crucial for interfaces to spatial databases. Two such aspects are treated in our system and are discussed below.

4.1 Eigenspace vs. Existential semantics

While previous work on prepositional semantics mainly dealt with relationships between two distinct objects, GIS queries often correspond to relationships between sets of objects. Consider the following query: "Buildings that are up to 200m from a lake". In case there is more than one lake, we expect our query to return any building such that there is at least one lake up to 200m from it. In other words, it appears like the query existentially quantifies over the lakes. On the other hand, if we change our query to "Buildings that are at least 200m from a lake", we would expect the query to return buildings that are over 200m away from *all* the lakes. The query "building that are between 200m and 500m from a lake" has a yet more complex semantics, and should result in any building such that there is at least one lake less than 500m from it and there is *no* lake less than 200m from it.

The semantics of the above three queries becomes much clearer, however, when instead of interpreting the indefinite "a lake" as an existential quantifier over the lakes in the database, "a lake" is interpreted as the set of all lakes, and distance is measured with respect to the space taken by the union of all lakes. We refer to this kind of interpretation for indefinites as *eigenspace semantics*. In SQL, the eigenspace of a set of objects is evaluated by using the aggregate function *GeomUnion* over a set of objects, as in:

Example SELECT geomunion(x.the_geom) FROM lake AS x;

In our framework, eigenspace semantics is treated by enabling a type-shifting from an indefinite noun-phrase into a special category G used for representing the eigenspace. The λSQL expression for G/N type-shifting is: $\lambda n.geomunion(\lambda g.\exists x.(n(x)\ AND\ g = x.the\_geom))$ where *the_geom* is the attribute for the geometrical data of an object in GIS database. The λSQL expression for $from$ $(((N\backslash N)\backslash RS)/G)$ is then defined as: $\lambda g.\lambda p.\lambda n.\lambda x.$ $(n(x)\ AND\ p(distance(x.the\_geom, g)))$.

It is important to note that while eigenspace semantics are used for spatial prepositions, in the case of other spatial relations that are not expressed using prepositions, such as the verbs *contains* and *intersects*, an indefinite is treated in the usual way, as an existential quantifier. For example, if we ask about "towns that contain a building with more than 10 floors", the eigenspace semantics would mean finding a town than contains *all* buildings with more than one floor, whereas we expect to get any town that contains at least one building

with more than 10 floors. We achieve the correct semantics in this case by providing a λSQL expression for verbs such as *contains* that existentially quantifies over the set of contained objects: $\lambda n1.\lambda n2.\lambda x.\exists y.(n1(y)\ AND\ n2(y)\ AND\ contains(x.the\_geom, y.the\_geom))$.

4.2 Semantics of between

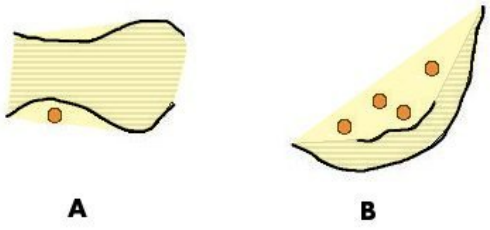

Fig 1. Example for between

Fig 2. Query result in QGIS

An additional aspect of spatial relations that was so far ignored in the semantic literature is relations between non-convex objects. A fundamental spatial relation which is quite problematic in the context of non-convex objects is the 3-place relation *between*.

Zwarts and Winter [18] suggest the following definition for *between*: X is between Y and Z if $X \subseteq convexHull(Y \cup Z) \setminus Y \setminus Z$, for convex objects in X, Y and Z. The problem is that many objects we deal with in the context of GISs are not convex. For example, it could be quite handy to talk about objects between two streets. However, streets are often non-convex shapes. As can be seen in figure 1, the convex hull for two streets represented by the solid lines includes areas that do not agree with our understanding of the expression *between the two streets*. In order to overcome this problem, we suggest the following definition:

Definition 4.1 Let X, Y and Z be sets of points. We say that X is between Y and Z iff either there is a point x on the border of Y such that the shortest line connecting x to Z crosses X, but does not cross Y, or there is a point y on the border of Z such that the shortest line connecting y to Y cross X, but does not cross Z.

The areas between the streets according to Definition 1 are marked by stripes. As can be seen from the illustration, the new definition is more in agreement with our intuitive understanding of *between*. Note that while the above is a strict definition of *between*, in some context people might use a sloppy definition (e.g., Buxton is between Manchester and Sheffield). In our system, however, we wish to avoid the vagueness of such sloppy definitions.

5 Implementation

The NLI presented in this paper was implemented in C++. The parser reads lexicon from a text file that includes the syntactic categories, and the semantics is represented using λSQL expressions for all data-independent lexical items. Data-dependent items are represented using templates, as explained in section 3.3. When the user enters a natural-language query, the query is parsed using a bottom-up right-to-left tabular Combinatorial Categorial Grammar (CCG) parser that was developed as part of the NLI prototype. The resulting λSQL expression is then converted into an SQL query as explained in section 2, which is sent to a spatially enabled database engine.

The system presented here uses PostGIS (http://postgis.refractions.net/) as a back-end. PostGIS is an open-source GIS extension to the PostgreSQL database engine, which implements the OpenGIS "Simple features specification for SQL" standard [2]. PostGIS basically supplies a set of functions that operate on vector representations, such as a function that calculates distance between polygons. The SQL queries are sent to PostGIS, which generates the result in a form of a table which is loaded into a GIS front-end that supports PostGIS, such as QGIS (http://www.qgis.org).

For example, the query "Buildings that are up to 500m from the intersection of Elm street and Oak street" are converted into the SQL query, which generates the result in figure 2:

(SELECT x.* FROM building AS x WHERE distance(x.the_geom, intersection((SELECT GeomUnion(x2.the_geom) FROM street AS x2 WHERE x2.street_nam LIKE 'elm'),(SELECT GeomUnion(x3.the_geom) FROM street AS x3 WHERE x3.street_nam LIKE 'oak')))<=500)

6 Conclusions and future work

This work presents an interface to GISs that is based on a controlled natural language. It demonstrates that it is possible to build such usable interfaces and express quite complex queries using a fragment of English. Future work on this subject can be done on several different levels: expanding the lexicon further by adding quantifiers, comparison between attributes of different objects and possibly anaphoric expressions. More thorough theoretical study is required regarding semantic issues such as eigenspace and *between* presented here, and finally, an empirical study is necessary to evaluate how usable such interfaces are for actual GIS users of varying skills and needs. We believe, however, that the general architecture and prototype demo interface that we suggest can be developed into a useful tool for planners and other professional users of GISs.

References

[1] I. Androutsopoulos and G. Ritchie. Database interfaces. In Dale, Moisl, and Somers, editors, *Handbook of Natural Language Processing*, chapter 9, pages 209–240. Marcel Dekker Inc., 2000.

[2] Open Geospatial Consortium. *Simple Features Specification for SQL.*

[3] M. Egenhofer and J. Herring. Categorizing binary topological relations between regions, lines and points in geographic databases. Technical report, Department of Surveying Engineering, University of Maine, Orono, ME, 1991.

[4] P.P. Filipe and N.J. Mamede. Databases and natural language interfaces. In *JISBD 2000*, pages 321–332, 2000.

[5] A.U. Frank and D.M. Mark. Language issues for GIS. In D. MacGuire, M.F. Goodchild, and D. Rhind, editors, *Geographical Information Systems: Principles and Applications*, pages 147–163. Wiley, New York, 1991.

[6] A. Hershkovits. *Language and Spatial Cognition: an interdisciplinary study of the prepositions in English.* Cambridge University Press, Cambridge, 1986.

[7] M. Kracht. On the semantics of locatives. *Linguistics and Philosophy*, 25:157–232, 2002.

[8] D.M Mark, S. Svorou, and D. Zubin. Spatial terms and spatial concepts: Geographic, cognitive and linguistic perspectives. In *International Geographic Information Systems (IGIS)*, pages 101–112, Arlington, VA, 1987.

[9] M. Minock. A phrasal approach to natural language interfaces over databases. In *NLDB-2005*, Alicante, Spain, June 2005.

[10] R. Nelken and N. Francez. Querying temporal databases using controlled natural language. In *18th conference on Computational linguistics - Volume 2*, pages 1076–1080, 2000.

[11] I. Pratt. Temporal preposisions and their logic. *Artificial Intelligence*, 166(1–2):1–36, 2005.

[12] Mukesh Kumar Rohil. Natural language processing to query a geographic information system(india) knowledgebase. In *Map India*, 2000.

[13] I. Schlaisich and M. Egenhofer. Multimodal spatial querying: What people sketch and talk about. In C. Stephanidis, editor, *1st International Conference on Universal Access in Human-Computer Interaction*, pages 732–736, New Orleans, LA, August 2001.

[14] J. Star and J. Estes. *Geographic Information System, An Introduction.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[15] Fangju Wang. Handling grammatical errors, ambiguity and impreciseness in GIS natural language queries. *Transactions in GIS*, 7(1):103–121, 2003.

[16] H. Wang, A.M MacEachren, and G. Cai. Design of human-GIS dialogue for communication of vague spatial concepts based on human communication framework. In *GIScience 2004*, Adelphi, MD, 2004.

[17] J.M. Zelle and R.J. Mooney. Learning to parse database queries using inductive logic programming. In *Thirteenth National Conference on Aritificial Intelligence*, pages 1050–1055, Portland, OR, August 1996.

[18] J. Zwarts and Y. Winter. Vector space semantics: a modeltheoretic analysis of locative prepositions. *Journal of Logic, Language and Information*, 9:171–213, 2000.