Making 1:N Explorable: a Search Interface for the ZAS Database of Clause-Embedding Predicates

Peter Meyer[1], **Thomas McFadden**[2]

[1]Institut für Deutsche Sprache, R 5, 6-13 D-68161 Mannheim
[2] Leibniz-Zentrum Allgemeine Sprachwissenschaft, Schützenstr. 18, D-10117 Berlin
E-mail: meyer@ids-mannheim.de, mcfadden@leibniz-zas.de

Abstract

We introduce a recently published corpus-based database of German clause-embedding predicates and present an innovative web application for exploring it. The application displays the predicates and the corpus examples for these predicates in two separate tables that can be browsed and searched in real time. While familiar web interface paradigms make it easy for users to get started, the data presentation and the interactive advanced search components for the two tables are designed to accomodate remarkably complex query needs without the need for resorting to a dedicated query language or a more specialized tool. The 1:n relationship between predicates and their examples is exploited in the two tables in that, e.g. the predicate table also shows, for each predicate and each example attribute, all values that occur in the examples for this predicate. An easy-to-use visual query builder for arbitrary boolean combinations of search criteria can optionally be displayed to pre-filter the underlying data presented in both tables. Several options for altering quantifier scope can be activated with simple checkboxes and considerably widen the space of searchable constellations.

Keywords: user interface; lexical data; query building; relational database

1. Introduction

This paper discusses the conceptual underpinnings of a web application user interface for exploring a recently published multilingual corpus-based database of clause-embedding predicates (Stiebels et al., 2017: http://www.owid.de/plus/zasembed2017/main). The relational database represents two basic types of entities that stand in a 1:n relationship: predicates (mostly verbs) and the corpus examples selected for a given predicate. Each of the two types is annotated with its own set of attribute-value pairs (henceforth, example properties vs. predicate properties). In each set, some attributes only apply to a certain subset (e.g. the definiteness attribute only applies to examples with embedded nominalization).

Despite the simplicity of a 1:n relationship, different quantifier/negation scope constellations can give rise to remarkably complex search semantics. In view of this complexity, the user interface was designed with the following goals: to present the user with a simple initial tabular view of the data that has straightforward and easy-to-use real-time filtering and sorting options, in line with accepted search interface design principles (Hearst, 2009; Morville & Callender, 2010; Russell-Rose & Tate, 2012), and to empower advanced users to incrementally construct ever more complex queries without resorting to domain-specific or generic query languages or introducing a separate "advanced search" interface layer.

Section 2 gives a short overview of the linguistic background and history of the ZAS database and introduces its basic data structure. In Section 3, we discuss the requirements that this structure and the target audience impose on a sufficiently elaborate search tool and how this affects the general objectives of the web application. The design decisions that were made to meet these requirements are presented in-depth in Section 4, and we briefly sketch the front-end and back-end technology in Section 5. The closing Section 6 points out a number of limitations of the tool in its current state, discussing some alternatives and competing approaches.

2. The ZAS database of clause-embedding predicates

2.1 Background on the database

The ZAS database of clause-embedding predicates documents how lexical predicates interact with clausal complementation. The examples in (1) give a simple demonstration from English of the kinds of patterns that are of interest.

(1) a. Max believes/knows/hopes [that Sarah works there].
b. Max *believes/knows/*hopes [whether Sarah works there].
c. Max *believes/*knows/hopes [to work there]

While a finite declarative clause is possible as the complement of *believe*, *know* or *hope*, only *know* can introduce finite interrogatives, and only *hope* can take control infinitives. It is thus well known that the properties of specific lexical predicates are important for understanding clausal embedding. In much of the literature, the discussion of complementation types and their licensing has focused on a relatively small number of predicates, taken to be representative of large classes with the same behavior (e.g. *believe*-class vs. *try*-class vs. *want*-class verbs in the discussion of English infinitives). However, it is clear that this oversimplifies matters and fails to capture interesting variation and crucial differences in how specific predicates interact with their syntactic environment. An important demonstration of this point can be found in Levin (1993), which examines in detail the range of distinct classes that can be identified for English verbs based on the structural alternations they engage in.

The ZAS database grew out of the conviction that a similar level of attention to detail is necessary to understand clausal complementation and what properties of lexical predicates are relevant for the behavior of their complements (see also Stiebels, 2011: for more detailed discussion). The methodology chosen to tackle this problem was to build a research tool around an extensive collection of data, illustrating the types of embedded clauses found with a large number of lexical predicates. Whereas a common prior strategy has been to *assume* predicate classes based on external semantic criteria, and then to investigate their behavior, we wanted to make it possible to *identify* classes of predicates based on the properties of the clauses they embed. We would thus collect, for each predicate included, a series of examples of different types of embedding, annotating each for a number of relevant grammatical properties, with the goal of illustrating all of the grammatical embedding possibilities for each predicate.

The database was conceived and initiated by Barbara Stiebels and gradually built up and extended by a team of researchers and student assistants at the Leibniz-Zentrum Allgemeine Sprachwissenschaft (ZAS) in Berlin (http://www.zas-berlin.de) from 2003 onwards. After her 2012 departure for the University of Leipzig, coordination of the project was taken over by Thomas McFadden in 2014. Through most of this period, the focus has been on contemporary German, though significant data have been collected for a number of other languages and older stages of German. It was decided early on that the data should not come from invented examples, but should be naturally occurring sentences extracted from corpora. The two most important corpus sources for the contemporary German portion are the *Digitales Wörterbuch der Deutschen Sprache* (DWDS; http://www.dwds.de), and the *Deutsches Referenzkorpus* (DeReKO; http://

www1.ids-mannheim.de/kl/projekte/korpora/). The embedding types systematically collected are infinitival (2-a) and nominalized (2-b) complements, verb-final finite declarative (2-c) and interrogative (2-d) complements (both polar and *wh*-questions), and embedded verb-second clauses (2-e); coverage of parentheticals and direct speech complements is ongoing work.

(2) a. Der Vorsitzende befahl, den Zeugen aufzurufen
the chair ordered the witness to-call
'The chair ordered the witness to be called' (ZDB 1565: DWDS K-Ge 1910)

b. Sie müssen sich mit dem Verkauf der Wohnungen beeilen.
the must themselves with the sale of-the apartments hurry
'They need to rush the sale of the apartments.' (ZDB 1523: DWDS BZ 1995)

c. Wir machen ab, daß er mich um acht Uhr besucht.
we make off that he me at 8 o'clock visits
'We agree that he will meet me at 8 o'clock.' (ZDB 70: DWDS K-Be 1980)

d. Gib acht, was du dir wünschst!
give attention what you yourself wish
'Be careful what you wish for.' (ZDB 218: IDS wpd 2011)

e. Aber ich ahne, es wird nicht mehr als Blech.
but I suspect it becomes not more than sheet-metal
'But I can tell it's just going to be nonsense.' (ZDB 256: IDS brz 2006)

The idea is that every predicate included should be checked in all relevant meaning variants and valency patterns, with a series of properties relevant for specific complementation types checked systematically. For example, with predicates that can embed finite verb-second clauses like (2-e), we checked for both indicative and subjunctive examples, and with predicates that can embed control infinitives like (2-a), we searched for examples with different types of control. Every example was then coded for these and several additional relevant properties. The guiding principle is that only "surfacy" features should be coded in order to keep the annotation theory-neutral and operationalizable.

In mid-2014, a collaboration was initiated with Carolin Müller-Spitzer and Peter Meyer of the Institut für Deutsche Sprache (IDS) in Mannheim, with the goal of making a version of the database publicly available on the OWID^{plus} platform for lexical-lexicographic data and resources (http://www.owid.de/plus/). A new search interface designed specifically for the ZAS database was then developed by Meyer in consultation with the ZAS team. The current public beta release of the database, which is the focus of this paper, contains only the data from the contemporary German part of the database. Additions are planned, however, for future releases of data on other languages and the historical stages of German.

2.2 The structure of the database and the 1:n relationship

The ZAS-internal version of the database is implemented in MySQL, with an in-house interface written in PHP for entering, editing and searching in the data. It is built primarily around two sets of data and the connections between them: a table of predicates and a table of corpus examples. Each example is associated with one predicate — it demonstrates one particular embedding use of that predicate. The two tables consist of a series

of entries, each of which contains several pieces of information on a single predicate or example. An entry in the predicate table contains the (infinitival) form of the predicate itself, as well as information about its syntactic category and morphological make-up. An entry in the example table contains far more information, with values for up to a dozen properties. This includes the text of the example itself, an indication of the argument structure and realization of the matrix clause, finiteness and word-order properties of the embedded clause, what complementizer it is introduced by (if any), whether it is an interrogative, as well as information about the corpus source and a link to the entry for the predicate. As for the size, the contemporary German version currently contains data on over 1700 distinct lexical predicates, exemplified through nearly 17,000 naturally occurring corpus examples.

While there is some additional complexity in the internal implementation (e.g. to deal with multiple languages and allow for easy extensibility), the system's primary goal is to provide information about examples and predicates, and the public version of the database and the OWID^{plus} search interface are built around this idea. At any given time, the interface displays either an example table view or a predicate table view, and every search query is ultimately interpreted as a search for either predicates or examples fitting certain criteria. At a basic level, this is fairly straightforward, but there are some cases where the interactions between predicate properties and example properties can lead to significant complexity. This arises from the fact that, while each example is tied to exactly one predicate, a given predicate will normally be associated with several examples. Dealing with this 1:n relationship presents interesting challenges for the design of the search interface, and thus will be extensively discussed throughout this paper. To set the stage, it will be useful to go into a bit more detail here about how examples, predicates, and their respective properties interact in the structure of the database itself.

The database is designed to enable sophisticated searches in order to obtain lists of predicates with complex combinations of properties. But it is the examples that constitute the bulk of the data, collected and coded for research use in the database, and what primarily interests us about the predicates is what kinds of embedded clauses are found in the examples associated with them. The information about these clauses is recorded in example properties, and thus to a large extent we search for predicates not by specifying their own properties, but those of the examples they embed. For example, we might want to search for all predicates that can embed subjunctive verb-second clauses, but no infinitives. Of course, it works the other way around as well. When searching for examples, some of the properties we might be interested will actually be properties of the predicate. We could e.g. search for all examples where the embedding predicate is a denominal verb in the hopes of finding out whether this correlates with the control status of embedded infinitives.

The crucial thing is that, despite this parallel, example properties and predicate properties are logically distinct in the way they function, and this is precisely because of the 1:n relationship. Let us consider the predicate properties first, because their status is simpler. In a search for predicates, constraints on predicate properties filter the results in an obvious way. If we specify the value Pt-V for the predicate property "morphology", the search will only return predicates that are morphologically particle verbs. The handling of predicate properties is just as simple in a search for examples, because for every example, there is exactly one predicate. We can thus treat predicate properties as though they

were example properties: we can search for *examples* with the value Pt-V for the property "morphology" (in Section 4.3 we will introduce the term *derived example criterion* for this concept), and rather than a list of all predicates that are morphologically particle verbs, we will get a list of all examples in which the predicate is morphologically a particle verb.

This simple situation does not hold with example properties. Matters are straightforward in an example search, because constraints on example properties simply filter examples. If we specify the value KONJ I for the example property "verb mood", the search will return only examples in which the embedded clause is finite and in the subjunctive I mood. But in a predicate search, the logic of 1:n makes things much more interesting. Because each predicate is associated with many examples, we cannot simply translate an example property into an implicit predicate property. We cannot say "return all predicates with the value KONJ I for the example property "verb mood"", because there is no unique example associated with each predicate. Rather, the relationship between predicates and example properties is more complex and has to be made clear in the search. The basic idea is that you are searching for predicates which are associated with at least one example that is characterized by the example property. We can rephrase this in the terms of the example above as "return all predicates which appear in at least one example which has the value KONJ I for the example property "verb mood"". By itself, this step is not particularly difficult, but it raises interesting questions when it comes to building complex queries involving more than one property.

Imagine now that we are interested in both subjunctives and embedded verb second — two properties that have often been thought to go together in German. There are two different ways to understand a search for predicates that can embed subjunctive clauses and verb-second clauses. A simple conjunction of two queries might return all predicates that have at least one subjunctive I example and at least one verb-second example. In this case, subjunctive I and verb second are independent, and because each predicate is associated with multiple examples, they may both apply to the same example, like (3-a), or they may apply to distinct examples associated with a single predicate, like (3-b) which is subjunctive I and (3-c) which is verb second.

(3) a. Er droht an, er werde nun jemanden befragen.
he threatens he will.SBJI now someone question
'He's threatening that he'll question someone now.' (ZDB 356: DWDS K-Be 1999)

b. Man nahm an, daß Leben ohne Licht unmöglich sei.
one took on that life without light impossible be.SBJI
'It was assumed that life was impossible without light.' (ZDB 624: DWDS TS 1999)

c. Zdenka hat sich ihrerseits in Matteo verliebt und schreibt ihm die
Zdenka has herself her-side in Metteo loved and writes him the
Liebesbriefe, von denen er annimmt, sie stammen von Arabella.
love-letters from which he assumes they come from Arabella
'Zdenka for her part has fallen in love with Matteo and writes him love letters, which he assumes come from Arabella.' (ZDB 629: DWDS K-Wi 1998)

This may indeed be what we want. But a different possibility is that we are interested in predicates that can embed a subjunctive I verb-second clause, i.e. we want single examples like (3-a) in which both properties hold. A task for our tools is thus to make both of these logical combinations of multiple example properties in a single predicate search possible in a way that is as easy as possible to understand. We will discuss the way the interface does this in Section 4.3.

3. Requirements for the UI

The new user interface for the published version of the ZAS database on the OWID^{plus} platform has to meet two broad requirements which we will discuss in turn. First, it has to provide facilities for formulating queries that can take full advantage of the range of data stored in the database and the connections between them. Second, it must be useable, at least at a basic level, for researchers who are interested in the behavior of clause-embedding predicates but have limited experience with databases and sophisticated search tools.

3.1 Semantic requirements for possible searches

The minimum capabilities necessary for the interface to actually reflect the structure of the underlying database are to allow search queries for both examples and predicates, where both types of query can refer to both example criteria and predicate criteria. To really exploit the full capabilities of the database, the interface should additionally provide the means to build complex queries combining multiple example and predicate criteria. The simplest form of this would be to allow the conjunction of criteria, interpreted so that the results returned by a search would be the examples or predicates simultaneously meeting all of the criteria specified. We discuss how the interface manages this in sections 4.1 and 4.2. The 1:n relationship means, however, that even this simple conjunction can potentially involve distinct semantics when a search for predicates involves more than one example criterion. One could design an interface that allows searches with all arbitrary boolean combinations of the different types of criteria, but much of this complexity is unlikely to be particularly useful for the study of lexical effects on clausal embedding, and certain types of combinations are more likely to lead to searches that are difficult to interpret than to allow the posing of typical research questions. We discuss the relevant trade-offs and the design decisions made in Section 4.3.

3.2 UI design and usability requirements

The original ZAS-internal interface was designed for team members working *on* the database. It thus includes facilities for entering and editing data in addition to running searches, and it works on the assumption that users are well acquainted with the structure and workings of the database. The new interface for the public version, however, is intended purely as a way to search and explore the database, for a wide audience of users, including novices. Thus the following considerations guided the design process:

- The interface should be explorable and discoverable for users with various degrees of prior experience.
- It should present an intuitive view of the data, making use of established interface design metaphors familiar from other web applications so that users can easily understand what they are looking at and how they can manipulate it.

- The view should be updated in real time whenever the user takes any action, so that they get immediate feedback and can quickly explore the consequences of different types of input.
- Running a basic query for an example or predicate satisfying some criterion should be extremely easy.
- Running complex queries involving boolean combinations of several criteria, while differentiating the various semantics relating to the 1:n relationship, should be possible.
- Ideally, it should be possible to get from the simplest search to the most complex by adding pieces step-by-step, where each intermediate step is a valid query, so that users can build their way up from novice to expert usage.
- The system should be equipped with extensive documentation, with facilities for accessing relevant parts directly from specific bits of the interface.

4. Design of the UI

4.1 Central concepts and basic search

The design of the user interface reflects the fact that the database is structured around two tables. At any given time, a version of either the example table or the predicate table is presented. The rows correspond to distinct entries — predicates or examples — and the columns display the properties associated with each entry. Every type of user input operates on one of these two views, with the results shown by updating the view in real time. Entering and modifying search queries manipulates restrictions on the entries displayed in the table. Thus there is no dedicated "query entry view" or "search results view", but a single view combining both, allowing users to immediately see the effects of the search criteria they enter and to modify them on the fly in order to test out and craft precise queries.

The two tables are identical in how they work and respond to input and boast the same system of integrated documentation — a detailed User Guide with ⓘ markers adjacent to the various interface elements that link directly to the relevant section of the Guide. The interface also provides facilities for exporting the data currently displayed in either table for local storage and processing. This function is disabled in the current public beta but will be activated in future releases. Both tables also allow for an "advanced search" which adds a more sophisticated query builder to the usual table. Note, however, that the advanced search is not an alternative to the basic table views, but rather an extension. The full functionality of the basic example and predicate tables is still available and works in the same way, just with additional possibilities for filtering the data. This will be described in detail in Section 4.2.

Switching between the example and predicate tables of course radically alters the way in which the data are presented. Even still, in most cases this change does not actually affect which data are presented, only the perspective from which they are viewed. This is because both table views combine information on both examples and predicates and display them in a single table, so that in general the same data can be presented either way — we say that the two tables are 'in sync'. The example table also contains predicate properties, since each example is associated with a predicate, and the predicate table also contains example properties, since each predicate is exemplified by a series of examples. There are, however, circumstances in which the tables can go out of sync, in particular

when searching for predicates based on example properties that can in principle apply to more than one example. This will be discussed in Section 4.3.

The main properties of the basic table views can be seen in Figure 1, with the relevant numbered features explained below.

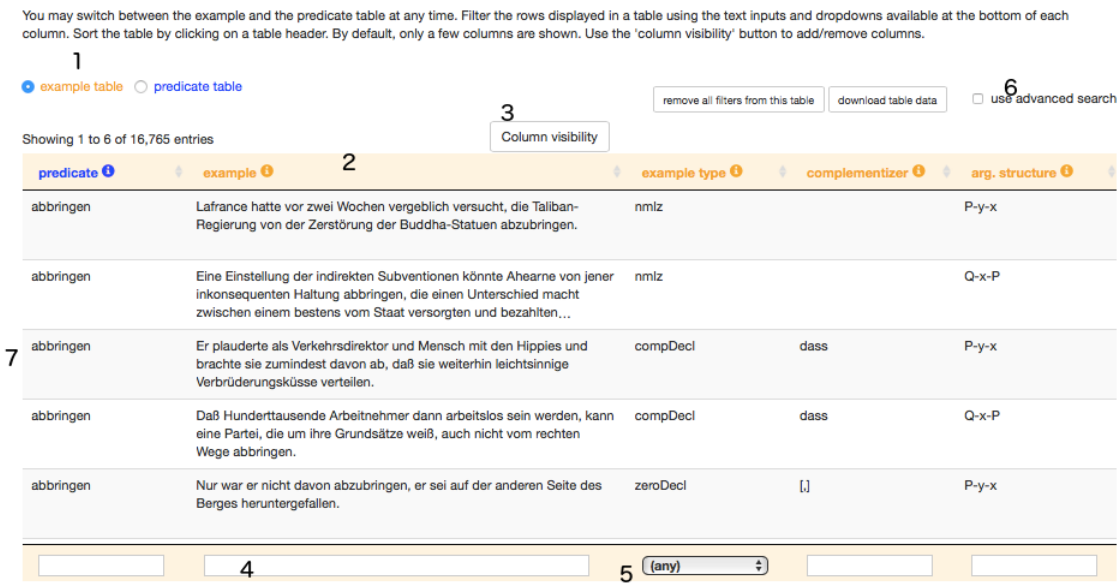

Figure 1: Basic Search

1 Selection of either the example or the predicate table view
2 Headers showing the properties currently displayed, blue for predicate properties, orange for example properties. Clicking on a property sorts the table by its values.
3 Facilities for specifying which properties are displayed as columns
4 Text box for entering a string that should be contained in the value for the relevant property, with autosuggest functionality and regular expressions
5 Pull down, for properties with a small number of permissible values
6 Click here to build an advanced search
7 List of entries in the table that match the current search, updated in real time. Double-clicking a row brings up complete information on its entry.

4.2 Advanced searches with the query builder

The per-column filtering options of the two tables are good for simple, quick and intuitive searches, but they are restricted in the following ways:

- For each example or predicate property, at most one search criterion may be formulated.
- The search criteria cannot be negated.
- The only way the table filtering criteria can be combined is by logical conjunction, such that all criteria must be fulfilled at the same time.

The interface's advanced search option provides an additional layer of search functionality that eliminates these restrictions. The advanced search is not a separate mode of access, i.e. it does not replace the interactive and explorative data presentation in the two tables, but complements it by letting the user restrict the underlying data set that is presented. To be more precise, both tables present the search result for the advanced query, albeit from two different perspectives. The data may then undergo filtering and sorting in a table, which amounts to the logical conjunction of the advanced search criteria and the filtering criteria defined in a specific table. In a sense, the advanced query acts as an additional "super-filter" on both tables.

Advanced search can be activated and deactivated at any time by a simple mouse click. When activated, the advanced query builder is shown above the table. As with the standard table filters, any change a user makes to the advanced query is immediately reflected in both tables. The query builder component itself follows the design of search components in modern operating systems, such as the advanced search offered in the default file manager "Finder" on Apple computers or in the query builder integrated in Microsoft Outlook. It allows the user to formulate an arbitrary number of criteria, even multiple criteria concerning the same property. To this end, the user may add any number of *criterion selectors* using the "+" button. Each criterion selector offers all types of search criteria also available as table filters. Criterion selectors for example properties have the orange background of the example table, while those for predicate properties have the blue one of the predicate table.

Arbitrary boolean combinations are supported as follows: All criterion selectors have an additional drop-down menu for optional negation; in addition, there is a special type of search criterion called "group of conditions" that opens up a *subgroup* of search criterion selectors connected by possibly negated conjunction or disjunction (logical "or", "and", "nor" or "nand"), yielding four types of logical connectivity: "all/none/at least one/not all subgroup criteria is/are true". Subgroups may be nested inside subgroups to any depth. The top-level criterion selectors of the advanced search can be thought of as implicitly contained in a conjunction group. Figure 2 shows a complex advanced search with nested subgroups, yielding examples with a predicate containing the string "sag" that embed an infinitive clause or a subordinate clause with both a complementizer containing "d" and a verb in the subjunctive I.

The table-specific filtering options and the advanced search system show intentional overlap with regard to both functionality and design. Input widgets in tables and in advanced search work exactly the same way. When working with a specific table, the user can freely choose between adding a filter to the table or adding the same condition as an advanced constraint on the top-level of the advanced query builder.

The advanced settings for search semantics, to which we now turn, are somewhat different.

4.3 Advanced search semantics

As discussed above, the 1:n relation between predicates and examples gives rise to potentially complex search questions beyond mere boolean combinations of criteria. The user interface introduced in this paper offers a principled approach to altering the semantics of queries through three user-defined settings. A deeper understanding of these settings

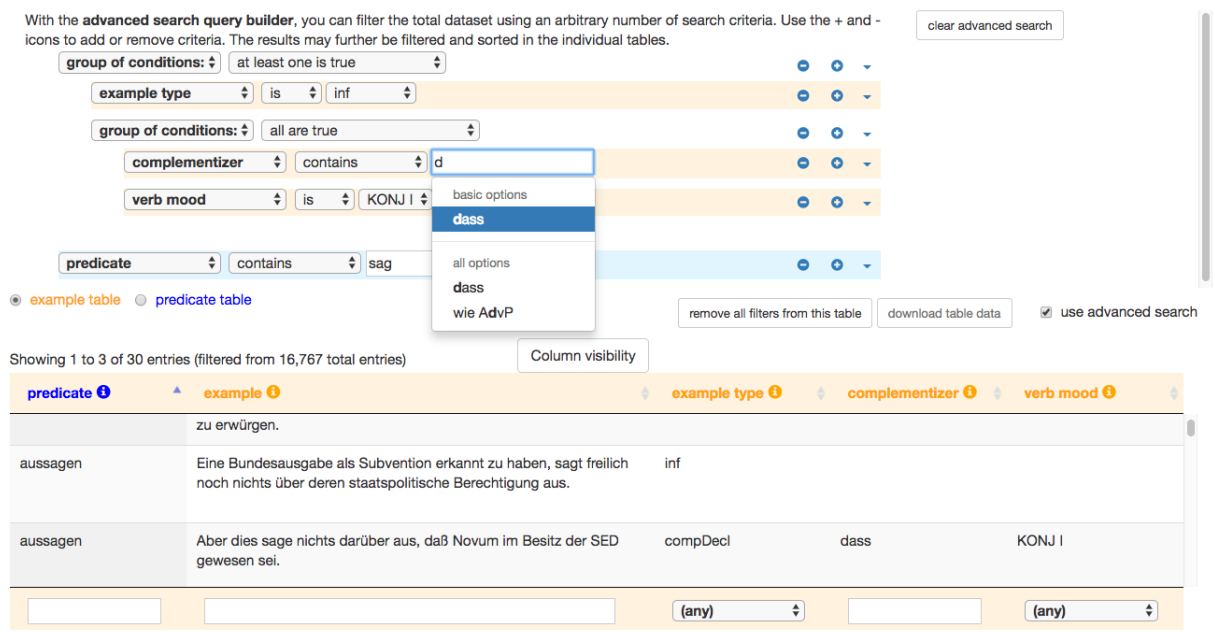

Figure 2: Building an advanced query

requires looking into how aspects of scope and negation interact in complex queries. A tutorial-style and hands-on introduction to these advanced aspects can be found in the online User's Guide. In what follows, we approach the subject from a formal perspective.

Let $\mathbb{E}$ denote the set of examples and $\mathbb{P}$ the set of predicates. Different example criteria will be denoted by E_1, E_2, etc., different predicate criteria by P_1, P_2, etc. An example criterion in the narrow sense (henceforth, *basic example criterion*) can only be applied to examples and puts a user-defined constraint on a specific example property; if the criterion E_1 actually applies to example $e \in \mathbb{E}$, this will be written in predicate-logic fashion as $E_1(e)$. A basic example criterion corresponds to an example property filter in the example table. Correspondingly, a predicate criterion in the narrow sense (henceforth, *basic predicate criterion*) can only be applied to predicates and puts a user-defined constraint on a specific predicate property; if the criterion P_1 actually applies to predicate $p \in \mathbb{P}$, this will be written as $P_1(p)$. A basic predicate criterion corresponds to a predicate property filter in the example table. Almost all search criteria can be used in negated form; we will use the superscript bar, as in $\overline{X}$, to signal user-defined negation of a criterion X, such that $\overline{X}(x) \Leftrightarrow \neg X(x)$. We write $pred(e)$ to denote the predicate exemplified by example e. To make the formulas a bit shorter and more legible, the letters e and p will always be used in such a way that $e \in \mathbb{E}$ and $p \in \mathbb{P}$ are implied.

In what follows, we assume that a search query is formulated in the advanced search query builder and we pose the question of how exactly these criteria define a search result set with respect to the two tables.

Queries concerning the example table are the easier part of the picture. For each predicate criterion P_j we define a *derived* example criterion E^{P_j} such that $E^{P_j}(e)$ is true iff $P_j(pred(e))$. If P_j, for instance, means "is a verb", then E^{P_j} stands for "is an example whose predicate is a verb". A derived example criterion corresponds to a predicate property filter in the example table. A single search criterion (regardless of whether we are dealing with table filters or with advanced query criteria) as applied to the exam-

ple table is either a basic or a derived example criterion. Since the example table shows the result of applying the conjunction of table filters and advanced search criteria to the entire database data set, searching the example table always means applying a boolean combination of basic and derived example criteria. To simplify formal exposition, we will stick to a sample advanced query consisting of a conjunction of two basic example criteria E_1 and E_2 and one basic predicate criterion P_3; extension to the general case is straightforward. When applied to the example table, an example e is shown in the table if and only if $E_1(e) \wedge E_2(e) \wedge E^{P_3}(e)$, which is equivalent to formula (1):

$$E_1(e) \wedge E_2(e) \wedge P_3(pred(e)) \tag{1}$$

Figure 3 shows a concrete case of this kind of query on the example table, in a search for examples with a particle verb (Pt-V) [criterion P] that embed an interrogative clause [criterion E_1] with a verb in subjunctive I mood [criterion E_2].

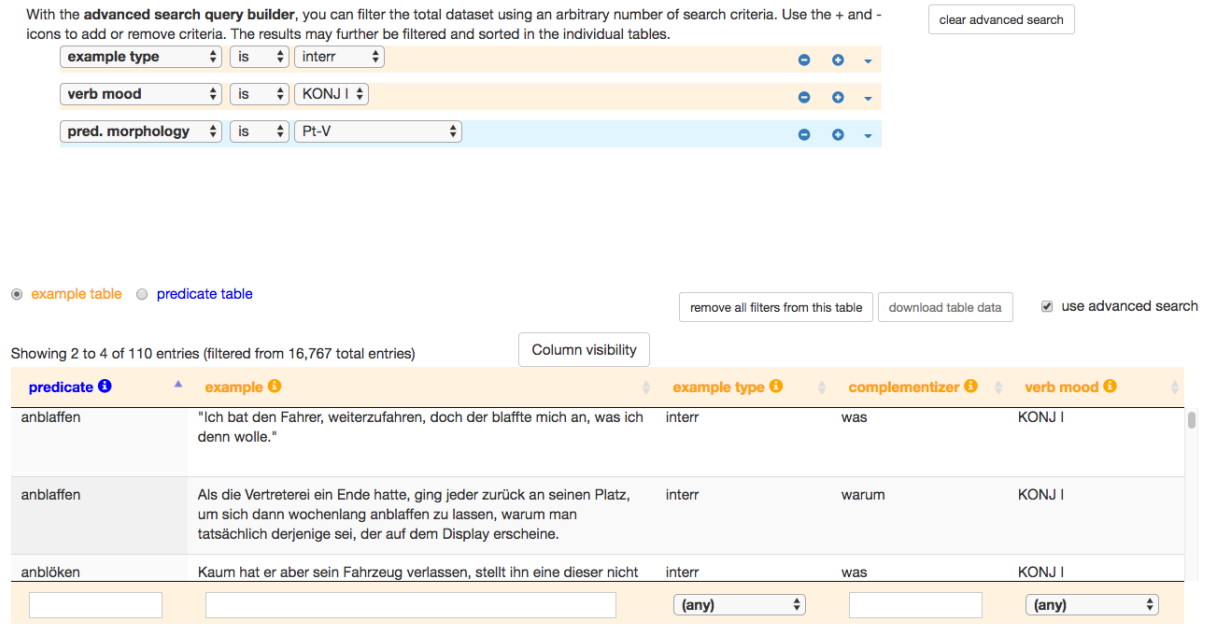

Figure 3: Sample advanced query with results in the example table

At this point, we will briefly discuss the semantics of example criteria derived from *negated* predicate criteria. The relevance of this interlude will emerge later. It is easy to see that $\overline{E^{P_j}}(e) \Leftrightarrow E^{\overline{P_j}}(e)$. In our previous example case, $\overline{P_j}$ would mean "is not a verb"; correspondingly, $E^{\overline{P_j}}$ represents the property "is an example whose predicate is *not* a verb", which is trivially coextensive with $\overline{E^{P_j}}$ "is *not* an example whose predicate is a verb". In other words, it is not necessary to separately define derived example criteria for negated predicate criteria; one can always negate the derived criterion instead.

Let us now turn to the way searches apply to the predicate table. When our sample advanced query with two basic example criteria E_1 and E_2 and one basic predicate criterion P_3 is applied to the predicate table, then, with default settings, a predicate p is shown in the table if and only if the basic predicate criterion applies to p and there is at least one example e for predicate p such that *both* example criteria apply to e. Formally, it

is required that $\exists e\,(p = pred(e) \wedge E_1\,(e) \wedge E_2\,(e)) \wedge P_3\,(p)$, which is equivalent to formula (2):

$$\exists e\,(p = pred(e) \wedge E_1\,(e) \wedge E_2\,(e) \wedge P_3\,(pred\,(e))) \tag{2}$$

In other words, the default semantics for searches in the predicate table requires that all basic example criteria be met simultaneously by (at least) one example e for p. The reason why the default settings for predicate table searches are defined like this becomes apparent from a comparison of formulas (1) and (2). It is easy to see that, with our sample search, a predicate p will appear in the predicate table if and only if at least one example for p appears in the example table. We say that the two tables are *in sync* in this case; this is a formalization of the intuitive notion, mentioned earlier, that both tables represent the same underlying set of data. This is, of course, also the ultimate reason why it is legitimate to have one advanced search applying to two different tables. Figure 4 shows the results of applying the sample query of 3 on the predicate table, in a search for particle verb (Pt-V) predicates [criterion P] for which there is at least one example that embeds an interrogative clause [criterion E_1] with a verb in subjunctive I mood [criterion E_2]

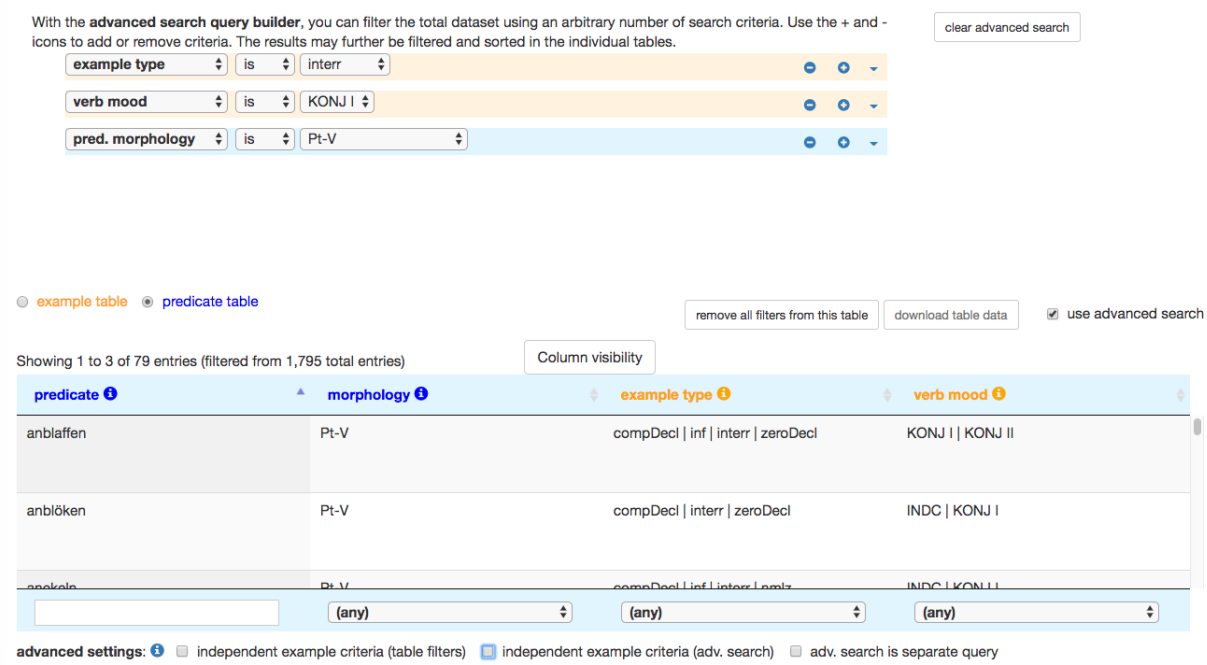

Figure 4: Sample advanced query with results in the predicate table

In many cases, the default semantics for the predicate table is not sufficient to meet users' needs. In our sample advanced query, a user might be interested in seeing all predicates p fulfilling criterion P_3 for which there is

- at least one example e_1 fulfilling criterion E_1 and
- at least one example e_2 (possibly, but not necessarily identical to e_1) fulfilling criterion E_2.

To handle this case, the user can specify what we (for want of a better term) call *independent example semantics* for the advanced query builder by ticking the "independent example criteria (adv. search)" checkbox appearing under the predicate table. With this semantics turned on, the search logic for the predicate table re-interprets the basic example criteria as *derived predicate criteria*. Clearly, deriving predicate criteria from example criteria has to be done differently than the other way around. We define, for each basic example criterion E_i, a *derived* predicate criterion P^{E_i} that holds of a predicate p iff $\exists e\,(p = pred\,(e) \wedge E_i\,(e))$. If E_i, for example, means "has an embedded infinitive clause", then P^{E_i} stands for "is a predicate with *at least one* example that has an embedded infinitive clause". With our sample query and independent example semantics turned on, a predicate p appears in the predicate table if and only if (3) holds:

$$P^{E_1}\,(p) \wedge P^{E_2}\,(p) \wedge P_3\,(p) \tag{3}$$

In formula (3), the two derived predicate criteria induce, by definition, two separate existential quantifications over the set $\mathbb{E}$ of examples, whereas the standard query semantics of (2) puts both example criteria in the scope of one existential quantifier. Figure 5 shows how the sample query of 3 is applied to the predicate table, this time with "independent example semantics", returning a list of particle verb (Pt-V) predicates [criterion P] for which there is at least one example that embeds an interrogative clause [criterion E_1] and at least one example with a verb in subjunctive I mood in the embedded clause [criterion E_2].

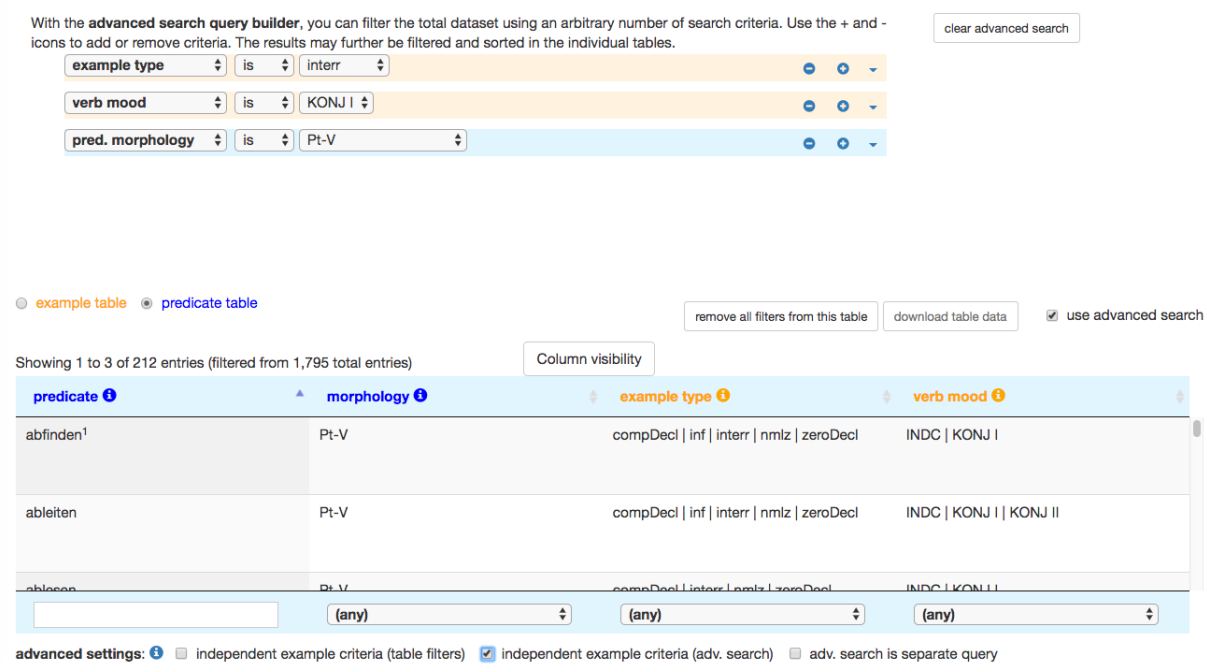

Figure 5: Sample advanced query with "independent example semantics"

The behavior of negation in derived predicate criteria is more complicated than with example criteria. It is easy to prove that $\overline{P^{E_i}}\,(p) \not\Leftrightarrow P^{\overline{E_i}}\,(p)$. In our previous example, $\overline{E_i}$ would mean "does not have an embedded infinitive clause"; correspondingly, $P^{\overline{E_i}}$ represents the property "is a predicate with at least one example *not* embedding an infinitive clause",

which is obviously not the same as $\overline{P^{E_i}}$ "is *not* a predicate with at least one example embedding an infinitive clause". This implies that P^{E_i}, $P^{\overline{E_i}}$, $\overline{P^{E_i}}$ and $\overline{P^{\overline{E_i}}}$ are, in general, four different criteria, because we have two logically different levels of negation. As far as the user interface is concerned, this implies that, for every derived predicate search criterion, two separate negation options would be needed to cover all possible cases. We decided to only offer one of these negation options: negating an example criterion always means negating the predicate criterion derived from it, as this seems to be the more intuitive and linguistically more relevant choice. In particular, it makes the formulation of queries such as the one in Figure 6 more plausible: With independent example semantics, this

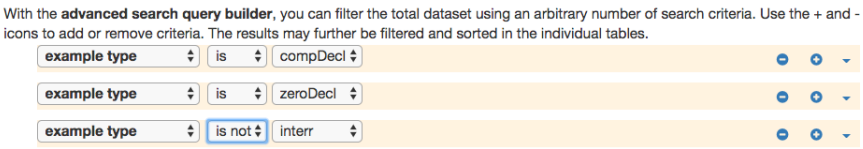

Figure 6: Advanced query with three example criteria, one of which is negated

query makes the system look for predicates whose examples exhibit the example types **compDecl** and **zeroDecl**, but not **interr**. If the negation on the third criterion were to be interpreted as pertaining to the underlying example criterion, then the query would read as follows: "Look for predicates that have at least one example with example type **compDecl**, at least one example with example type **zeroDecl**, and at least one example where the example type is not **interr**." Obviously, the third criterion would be redundant in this interpretation. Overall, the design of the system ensures that the conjunction of a criterion and its negation always yields an empty result set.

A major complication with independent example semantics is the fact that it puts the two tables "out of sync"; that is, they do not represent answers to the same query anymore. The query of Figure 6 produces zero results in the example table since no single example can fulfill all three conditions at the same time.

Independent example semantics can also be chosen for predicate table filters with the "independent example criteria (table filters)" checkbox, such that this semantics can be turned on and off separately for the two search components of the interface. If "independent example semantics" is activated neither for the advanced query builder nor for the table filters, all user-defined example criteria of both search components are, by default, in the scope of the existential quantifier of formula (2). This can be changed through a third checkbox "adv. search is separate query". If this setting is activated, the two components (criteria in the table filters vs. in the advanced builder) are treated separately and generate two separate searches according to formula (2). The two formulas are then joined with logical AND, returning the intersection of the two result sets. This is useful if a user looks for examples for predicates that have examples with multiple example properties A, B, C, ... and (possibly different) examples with multiple example properties D, E, F, ... Figure 7 shows the "separate query" setting in a query for predicates that can embed subjunctive 1 interrogative clauses and subjunctive II finite declarative clauses without a complementizer.

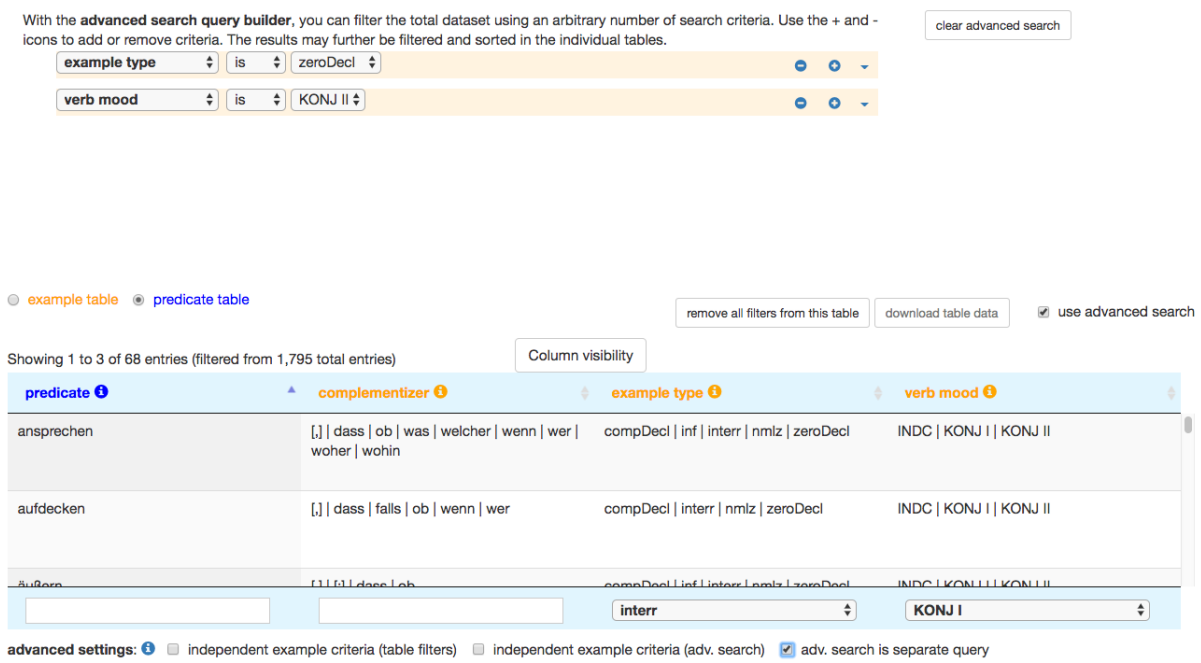

Figure 7: Advanced query for predicates with "separate query" setting turned on

5. Software architecture

Here we briefly sketch the software and data modeling strategy used to ensure that even complex search results can be presented in the form of potentially very long tables in real time in the browser, instantaneously adapting to every change a user makes in a search criterion. Response times have to be very short as each change in one of the search components, such as adding or deleting a single letter in a text field, generates a new server request.

5.1 Backend and database design

The version of the ZAS database of clause-embedding predicates published on OWID^{plus} is a self-contained web application running in a standard Java Servlet container (based on the Sparkjava framework, sparkjava.com) with an embedded relational database (H2, h2database.com). For the purposes of the online version, a snapshot of the 14 original MySQL database tables is systematically denormalized to construct a database of just two flat tables (examples vs. predicates) where each attribute, including each "inherited" property, is represented as a separate column, very similar to what is actually presented to the users in the interface. This "pivoting" procedure, though not strictly necessary, greatly reduces programming and execution overhead and minimizes the need for joins in SQL queries.

5.2 Frontend (browser) technology

An important feature of the user interface is that all data is presented in the form of scrollable tables. Loading up to 17,000 rows with more than 15 columns would take too long, however. Our solution to this problem consists in virtualizing the table (in our case, using the DataTables plugin, datatables.net). Only those data that are currently visible

in the browser (plus some spare rows) are loaded from the server; on scrolling, further rows are fetched using AJAX (XHR) requests.

5.3 Evaluation

For the amount of data available in the ZAS database, the approach outlined above delivers satisfactory response times even for complex queries involving joins, regular expressions etc. Preliminary tests show that the application scales well only up to some 100,000 data rows in the example table. By changing to an in-memory database, this limit can be pushed considerably; however, datasets with millions or even billions of rows would require a more elaborate way of indexing the data and, possibly, limiting the application of regular expressions.

6. Discussion and prospects

The user interface presented in this paper attempts to strike a middle ground between the availability of complex search options and ease of use. The tool deliberately resorts to a powerful combination of two familiar and easily accessible types of interactive interface components, viz. tables with sorting and filtering options and hierarchical query builders. In addition, a set of three yes/no settings can be used to alter the behavior of scope and negation, resulting in an amazing range of possible searches. Through the concepts of inherited columns, derived search properties and the default in-sync setting of search semantics, the 1:n relation between the two tables is exploited as much as possible.

On the other hand, it is self-evident that the query system is not relationally complete in the sense of Codd (1972) and, a fortiori, not equivalent in expressive power to standard SQL. The discussion in subsections 4.2 and 4.3 already pointed to several areas where the range of possible queries could easily be extended. The possible enhancements listed below are under consideration for future versions of the interface.

- With "independent example semantics" turned on for the predicate table, the advanced search criterion input widgets for predicate criteria derived from example criteria could offer both kinds of negation mentioned in subsection 4.3, such as "{at least one | no} example: example type {is | is not} {compDecl | zeroDecl | ...}".
- Instead of having one global, all-or-nothing setting for "independent example semantics", the interface could offer a choice to activate this semantics (separate quantification over example set) for each individual example property, e.g. through a checkbox available on all predicate criteria widgets. The downside is that it would be easy to build advanced queries whose precise meaning is difficult to understand for human users (and therefore not likely to be useful for pursuing typical research questions).
- An even more general approach to multiple quantifications on the example set $\mathbb{E}$ in the predicate table would be to explicitly introduce a mechanism of "scope subgroups" in the query builder. All example criteria within a scope subgroup would be under the scope of a separate existential quantifier on $\mathbb{E}$. Interpreting such queries can, again, be a demanding task for inexperienced users. On the technical side, the more scope subgroups are defined in a query, the more SQL joins appear in the database query on the server side, possibly impairing performance.

Finally, we compare our tool against other approaches. A textbook strategy for online presentation of two tables in a one-to-many relation would be to show the two tables on different web pages and to take account of the relational character of the data in the following way:

- create hyperlinks on the 'many' side (in our case linking the predicates mentioned in the example table to the corresponding row in the predicate table);
- create a master-detail view option on the 'one' side (in our case showing all examples for a given predicate upon, e.g., double-clicking a row in the predicate table).

Our solution does provide master-detail views for both tables, but interweaves both data presentation and search options for the tables to a much higher degree: each table includes as much information from the other table as possible; standard advanced search works in a cross-table way; "independent example semantics" options give more search power.

At the other end of the spectrum, a full-blown visual query tool for relational databases could be used to provide the user with the full expressive power of modern SQL. An important early example of a relational query language with a graphical interface is *Query By Example* (Zloof, 1977; cf. Ramakrishnan & Gehrke, 2002: chapter 6, pp. 177ff.). The most widely known visual query system today is probably the one found in Microsoft Access; a large number of interface components and full-blown web applications work in a similar way. However, such a system would not be friendly for the casual user and has a much steeper learning curve than the immediate interaction with tables. The case of "independent example semantics" shows how quickly query formulation can get very abstract: for each example property included in a predicate search with this semantics, an additional join with the example table must be created, i.e. a new "instance" of the example table must be added to the visual representation.

7. Acknowledgements

The work of the second author was supported by the Bundesministerium für Bildung und Forschung (BMBF, Grant Nr. 01UG1411). We would like to thank Barbara Stiebels and the ZAS database team, in particular Kerstin Schwabe, Torgrim Solstad, Livia Sommer, Katarzyna Stoltmann, Noemi Geiger, Gediminas Schüppenhauer and Sybille Kiziltan.

8. References

Codd, E.F. (1972). Relational completeness of data base sublanguages. In R. Rustin (ed.) *Data Base Systems, Proceedings of 6th Courant Computer Science Symposium*. New York: Prentice-Hall, pp. 65–98.

Hearst, M.A. (2009). *Search User Interfaces*. Cambridge University Press, 1st edition.

Levin, B. (1993). *English Verb Classes and Alternations*. University of Chicago Press.

Morville, P. & Callender, J. (2010). *Search Patterns: Design for Discovery*. O'Reilly Media, 1 edition edition.

Ramakrishnan, R. & Gehrke, J. (2002). *Database Management Systems*. Mcgraw-Hill, 2nd edition edition.

Russell-Rose, T. & Tate, T. (2012). *Designing the Search Experience: The Information Architecture of Discovery*. Morgan Kaufmann.

Stiebels, B. (2011). Von den Herausforderungen des lexikalischen Reichtums. In V. der Geisteswissenschaftlichen Zentren Berlin e.V. (ed.) *Bericht über das Forschungsjahr 2010*. pp. 51–72.

Stiebels, B., McFadden, T., Schwabe, K., Solstad, T., Kellner, E., Sommer, L. & Stoltmann, K. (2017). ZAS Database of Clause-embedding Predicates, release 0.2 (Public Beta). In *OWID^{plus}*. Institut für Deutsche Sprache, Mannheim. URL http://www.owid.de/plus/zasembed2017.

Zloof, M.M. (1977). Query-by-example: A data base language. *IBM systems Journal*, 16(4), pp. 324–343.